Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure

Marwan Abi-Antoun

CMU-ISR-10-114
May 14, 2010

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich (Chair)
Brad A. Myers
William Scherlis
Nenad Medvidovic, U.S.C.

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported in part by NSF CAREER award CCF-0546550, DARPA contract HR00110710019, Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems," the U.S. Department of Defense, and the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsors.

Keywords: object diagram, object graph, runtime structure, runtime architecture, component-and-connector (C&C) view, execution architecture, architectural extraction, conformance analysis, conformance measurement, ownership types, ownership domains, static analysis, points-to analysis, communication integrity, reverse engineering

To the Alchemists of the World!

There was a language in the world that everyone understood, a language the boy had used throughout the time that he was trying to improve things at the shop. It was the language of enthusiasm, of things accomplished with love and purpose, and as part of a search for something believed in and desired.

The Alchemist—Paul Coelho (p. 64), translated by Alan R. Clarke

Abstract

A high-level architectural diagram of a system's organization can be useful during software evolution. Such a diagram is often missing and must be extracted from the code. Alternatively, an existing diagram may be inconsistent with the code, and must be analyzed for conformance with the implementation. One important notion of conformance, the *communication integrity* principle, stipulates that each component in the implementation may only communicate directly with the components to which it is connected in the architecture.

This dissertation proposes a novel approach, SCHOLIA*, to extract a hierarchical runtime architecture from an existing object-oriented system, and analyze communication integrity with a target architecture, entirely statically and using typecheckable ownership annotations.

Previous approaches to enforcing communication integrity have significant drawbacks: they either require radical language extensions that incorporate architectural constructs at the expense of severe implementation restrictions, mandate specialized architectural middleware, or use dynamic analyses that cannot check all possible executions.

The key contribution is a static points-to analysis to extract, from an annotated program, a global object graph that provides architectural abstraction by ownership hierarchy and by types, where architecturally significant objects appear near the top of the hierarchy and data structures are further down. Moreover, an extracted object graph is sound in two respects. First, each runtime object has exactly one representative in the object graph. Second, the object graph has edges that correspond to all possible runtime points-to relations between those objects.

Another analysis abstracts an object graph into a built runtime architecture. Then, a third analysis compares structurally the built architecture to a target, and analyzes communication integrity in the target architecture, without propagating low-level implementation objects into the target architecture. An evaluation on several real object-oriented systems showed that, in practice, SCHOLIA can be applied to an existing system while changing only annotations in the code, and that SCHOLIA can identify interesting structural differences between an existing implementation and its target architecture.

*SCHOLIA stands for static conformance checking of object-based structural views of architecture. *Scholia* are annotations which are inserted on the margin of an ancient manuscript.

Acknowledgments

No man can reveal to you nothing but that which already lies half-asleep in the dawning of your knowledge — Khalil Gibran

First and foremost, I want to thank my advisor, Professor Jonathan Aldrich, for being a great mentor, *tormentor*, and role model. Jonathan was equally good at giving big picture advice on career plans, revealing deep technical insights, or nitpicking details of inference rules and proofs. I could never thank him enough for the countless hours spent meeting, advising, emailing, reviewing various paper drafts, critiquing presentations, and proof-reading a voluminous dissertation.

I want to especially thank Professor Medvidovic (Neno) for awakening in me the interest in software architecture while at USC. Several years later, Neno and my parents were one of the few who supported my desire to go back to graduate school. Most of my work colleagues at the time thought I was crazy to leave a perfectly good job and take a big paycut to go lead the spartan life of a graduate student for several years. Neno, thank you also for forgiving me that I went to CMU instead of USC. Believe me, there were several dreary winter days in Pittsburgh when I longed for sunny southern California. But the draw of the family was still worth it.

Professor Myers was a great source of advice on issues related to software engineering tools, usability and usefulness, and a great sounding board. Professor Scherlis gave me excellent advice on how to position the approach and conduct a field study with real software and developers.

I would like to thank Nagi Nahas. The project on which we collaborated during my first year at CMU gave me the confidence of "beginner's luck" and ended up being reused throughout this dissertation. I also want to thank my long-term officemate Thomas for being such a helpful, pleasant person and putting up with me on those days after receiving conference paper rejections.

I would like to thank all of my supporters, my detractors, my supporters turned detractors, my detractors turned supporters, as well as those who were completely indifferent. You all know who you are. I would like to thank my family, for always letting me know in which of the above columns you were, and for always being there for me. My Dad, my Mom, my brother, my sisters and my most adorable nephews helped me retain my sanity and brought much happiness.

Finally, I thank the Computer Science Department for turning my final week at CMU working on my thesis into a veritable obstacle course: from moving the office phone line without warning, and in the middle of an important conversation with the advisor, at the end of which, each one assumed the other had hung up; leaving no printer in Doherty, as well as keeping a single working elevator in Wean, a building with hundreds of occupants. I got to finalize my dissertation and get a cardiovascular workout at the same time, running from the 4th floor of Doherty to the 8th floor of Wean, then downstairs to the 5th floor, which still had a printer! As the late Professor Randy Pausch put it best in his Final Lecture (which I had the privilege of attending in person), brick walls are there for a reason—they let us prove how badly we want things. I did it!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's runtime structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa." – (Gamma et al. 1994, p. 22)*

This dissertation proposes a novel approach, SCHOLIA[1], to extract statically a hierarchical runtime architecture from a program in a widely used object-oriented language[2], using annotations. If a target runtime architecture exists, SCHOLIA can also analyze, at compile time, communication integrity between the code and the intended architecture. At its core, SCHOLIA relies on a novel static analysis to extract a hierarchical object graph from an arbitrary object-oriented implementation. The extracted object graph provides architectural abstraction by ownership hierarchy and by types. Moreover, the object graph is *sound* in two respects. First, each runtime object has exactly one representative in the object graph. Second, the object graph has edges that correspond to all possible runtime points-to relations between those objects. The extraction analysis assumes that typecheckable ownership annotations provide minimally invasive hints about the architecture, instead of requiring developers to use a specialized framework or a new programming language. To analyze conformance, SCHOLIA compares the built and the designed architecture using a structural comparison for hierarchical architectural views that does not assume that view elements have unique identifiers. Finally, SCHOLIA's conformance analysis allows the designed architecture to be more abstract, and accounts for additional communication in the implementation without propagating low-level objects into the designed architecture.

[1] SCHOLIA stands for static conformance checking of object-based structural views of architecture. According to Wikipedia, *scholia* are annotations which are inserted on the margin of an ancient manuscript. The metaphor is that this approach supports existing legacy, i.e., ancient, object-oriented systems and uses annotations that other development tools can ignore.

[2] This dissertation mainly considers Java-like statically-typed general purpose object-oriented languages such as Java and C#, where each object is a Plain Old Java Object (POJO). This work does not specifically address dynamically typed languages, or Java programs that use aspect-oriented programming (AspectJ), component frameworks such as Enterprise Java Beans (EJB), etc.

1.1 Introduction

During software evolution, the most reliable and accurate description of a software system is its source code. However, high-level architectural diagrams of the system's organization are also very important. For instance, a diagram can help locate the components that must be modified, or estimate the magnitude of the impact of a change based on the dependencies among entities.

Often, such a diagram is missing, hence the need to *extract* one from the code. Alternatively, a diagram may exist but may be inconsistent with the code. As a result, taking an important decision on how to evolve a system based on an incorrect architectural diagram may lead to problems during the implementation of the changes, or the implemented system may not exhibit the desired architectural qualities. Hence, there is an important need to analyze the *conformance* of a target architecture with an implementation.

This chapter is organized as follows. Section 1.2 discusses object-oriented design diagrams. Section 1.3 discusses architectural views. Section 1.4 discusses the notion of architectural abstraction. Section 1.5 discusses architectural extraction. Section 1.6 discusses analyzing architectural conformance. Section 1.7 discusses the proposed approach, SCHOLIA. Section 1.8 summarizes the requirements of a solution. Section 1.9 lists the contributions of this dissertation. Section 1.10 concludes with a thesis statement and an outline for the rest of this document.

1.2 Object-Oriented Diagrams

Reverse engineering or architectural extraction can extract various complementary high-level views of a system. A view can focus on the static code structure, or on the runtime structure. Most previous reverse engineering research focuses on the code structure, while this dissertation improves on the state-of-the-art for extracting and analyzing the runtime structure of object-oriented systems.

1.2.1 Example

I illustrate by example the key differences between the code structure and the runtime structure using Aphyds, a system of 8,000 source lines of Java code (not counting the libraries used), first discussed by (Aldrich et al. 2002a). Aphyds is a pedagogical circuit layout application that an electrical engineering professor wrote for one of his classes. Students in the class are given the program with several key algorithms omitted, and are asked to code the algorithms as assignments.

The design of Aphyds follows a two-tiered Document-View architecture. The designed architecture (Fig. 1.1) shows tiers, components, and interactions between components. In this diagram, an edge represents a points-to relation. User interface components such as viewerUI are in the upper half of the diagram. A circuit and computational components, such as partitioner, are the lower half.

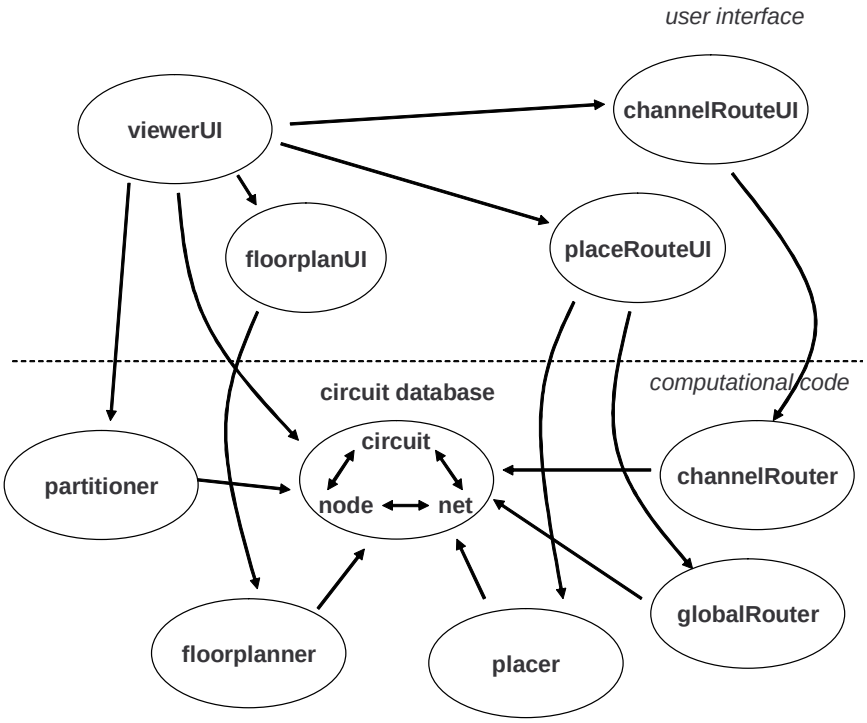


Figure 1.1: Aphyds: designed architecture, redrawn from the original developer's diagram reproduced in (Aldrich et al. 2002a), included here with some adaptations. I renamed some components, reversed the direction of some arrows (Aldrich et al. 2002a, p. 192) and excluded data flow edges since SCHOLIA does not currently show the latter.

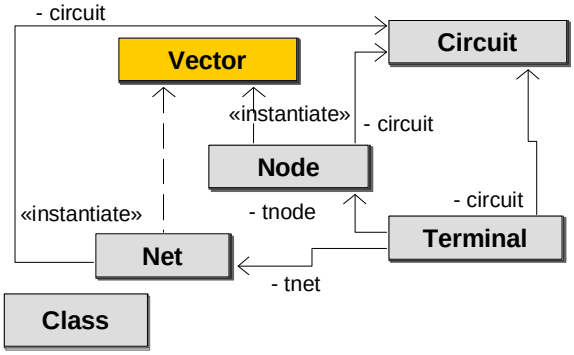


Figure 1.2: Aphyds: partial class diagram focusing on the class `Circuit` and related classes.

1.2.2 Class Diagrams

A *class diagram* is an important and widely used description of an object-oriented system that shows the static code structure, in terms of classes and fixed inheritance relationships.

Many tools automatically generate class diagrams of the code structure from program source (Kollman et al. 2002). I used the Eclipse UML tool (Omondo 2006) to extract a class diagram from the Aphyds code. For example, a class diagram would show one `Vector` class, and `Node` and `Net` classes that have a module dependency on `Vector` (Fig. 1.2).

1.2.3 Object Diagrams

Another important view is an *object diagram* or *object graph*, where nodes represent objects, i.e., instances of the classes in a class diagram, and where edges correspond to relations between objects. An object diagram makes explicit the structure of the objects instantiated by the program and their relations, facts that are only implicit in a class diagram. While in the class diagram a single node represents a class and summarizes the properties of all of its instances, an object diagram represents different instances as distinct nodes, with their own properties (Tonella and Potrich 2004). For example, (Gamma et al. 1994) used a class diagram and an object diagram to explain several of the standard design patterns. Recent empirical evidence confirms the importance of "how objects connect to each other at runtime when I want to understand code that is unknown: an object diagram is more interesting than a class diagram, as it expresses more how [the system] functions" (Lee et al. 2008).

1.2.3.1 Static vs. dynamic object diagrams

Following (Tonella and Potrich 2004), I distinguish between *static object diagrams* and *dynamic object diagrams*. A *static object diagram* represents all objects and inter-object relationships possibly created in a program, and is recovered by a *static analysis* over the code. A *dynamic object diagram* shows the objects and the relations that are created during one or more specific system executions, and is recovered using a *dynamic analysis*.

Static and dynamic object diagrams provide complementary information. A static object diagram lacks precision on the actual multiplicity of the objects that the program may create, or the actual relations between objects. In contrast, a dynamic object diagram, e.g., (Flanagan and Freund 2006), can show the exact number of instances and the actual relations in a given program run. But a dynamic object diagram may not reflect important objects or relations that show up only in other executions. For example, using a design diagram, a security review could enumerate all possible communication between trusted and untrusted parts of a system[3]. But if the diagram does not show all communication present in the implementation, because additional communication pathways arise during other executions, the analysis may be incorrect.

In general, there are several problems with dynamic analysis. First, a dynamic analysis may not include important objects or relations that show up only in other executions. Second, a dynamic analysis may not be repeatable, i.e., changing the inputs or executing different use cases might produce different results. Third, runtime heap information does not convey design intent. Fourth, a dynamic analysis cannot be used on an incomplete program still under development or to analyze a framework separately from a specific instantiation. Finally, some dynamic analyses carry a significant runtime overhead—a 10X-50X slowdown in one case (Flanagan and Freund 2006), which must be incurred each time the analysis is run.

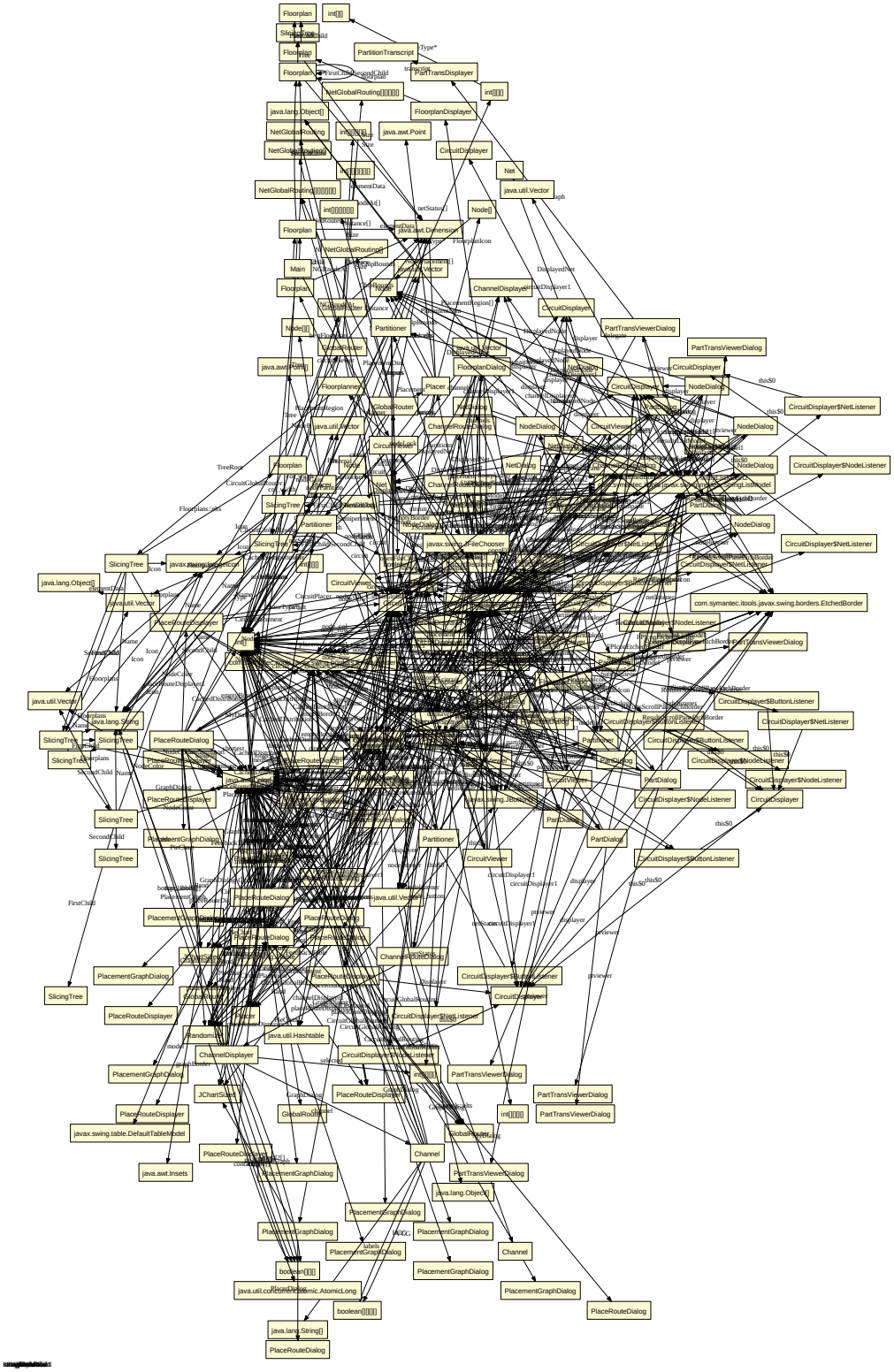


Figure 1.3: Aphyds: flat object graph, extracted statically by WOMBLE (Jackson and Waingold 2001). To read the labels, zoom in by 200%

1.2.3.2 Global object diagrams

Extracting a global object diagram that shows the entire application structure increases the diagram's complexity significantly. A flat object graph for an entire system often has a profusion of objects that makes it difficult to obtain a high-level picture, even for a relatively small program. For Aphyds, a flat object graph mixes low-level objects such as `SlicingTree` that are data structures, with architecturally-relevant objects such as `GlobalRouter` from the application domain, and a developer has no obvious way to distinguish between them (Fig. 1.3).

1.3 Software Architecture

In addition to object-oriented design diagrams, one can talk about the software architecture of a system. A software architecture is a high-level description of a software system that is a conceptual tool for documenting, reasoning about and communicating the structure of the system to developers or to other stakeholders. Different complementary architectural views describe a system from different perspectives (Soni et al. 1995; Kruchten 1995). In particular, there are two important architectural views, the *code architecture* and the *runtime architecture* that we discuss next. These views are the analogues of class diagrams and object diagrams, respectively.

1.3.1 Code Architecture

A *code architecture* or *module view* shows code entities in terms of classes, fixed inheritance relationships, packages, layers and modules (Clements et al. 2003). A code architecture impacts quality attributes like maintainability, and has mature tool support. For object-oriented code, a module view is often a class diagram or a package view. And today, many tools can extract such module views from code (Kollman et al. 2002).

1.3.1.1 Package (layer) vs. runtime tier

A code architecture often organizes classes according to their packages. However, an application's code package structure is often orthogonal to its runtime structure. For example, all the classes in Aphyds are in the same package. While this violates good programming practice, it highlights the difference between a code-level *package* or *layer* and a runtime *tier*[4].

A class diagram (Fig. 1.2) shows the classes `Circuit`, `Node`, `Net` and `Terminal` all at the same level. Of course, a class diagram can have hierarchy by using packages. But a package is just a namespace. Indeed, a developer often carefully designs the package structure to indicate her architectural intent. For instance, she may place the class `CircuitViewer` in the `aphyds.ui` package and the class `Partitioner` in the `aphyds.model` package, to indicate that `CircuitViewer` and `Partitioner` belong to different *layers* in the code architecture.

[3]Several companies have established a process for such security reviews, *threat modeling* (Howard and Lipner 2003; Torr 2005; Howard and Lipner 2006). Section 7.8.1 (Page 251) discusses threat modeling further.

[4]We adopt the terminology of (Clements et al. 2003): a *layer* denotes a cluster or a partition in a *code architecture* or a *module view*. A *tier* denotes a cluster or a partition in a *runtime architecture* or a *runtime view*.

A runtime architecture often groups conceptually related instances into conceptual runtime partitions or *tiers*. For instance, the Aphyds developer's diagram distinguishes presentation components such as viewerUI in the UI tier, from computational components such as partitioner in the MODEL tier.

In particular, the package which contains a class does not indicate to which architectural tiers the instances of that class belong. In the above case, the classes `CircuitViewer` and `Partitioner` from the packages `aphyds.ui` and `aphyds.model`, have their instances, viewerUI and partitioner, fit nicely within the `UI` and `MODEL` tiers, respectively. But, in general, one cannot represent the dynamic structure of an application using the static source code organization, because different instantiations of a class often have distinct conceptual purposes and correspond to different elements in the design. For example, the code would still have a single `Vector` class in a `java.util` package. But at runtime, the viewerUI component in the UI tier may have an instance of `Vector`, one that is different from a `Vector` instance that is in use by the partitioner component in the MODEL tier.

1.3.2 Runtime Architecture

Another architectural view, the *runtime architecture* or *runtime view*, models a software system as an organization of runtime entities, interactions between the entities, and constraints on how the entities interact. A runtime architecture is important, because it impacts quality attributes such as security, performance, and reliability.

Architecture description language (ADL). A runtime architecture can be an informal boxes-and-lines diagram, or a formal specification in an architecture description language (ADL) (Medvidovic and Taylor 2000). While many ADLs have been proposed, a common weakness of many ADLs is the lack of enforcement with an implementation.

SCHOLIA uses the Acme general purpose ADL (Garlan et al. 2000) to document the built and the designed architecture. Acme represents a hierarchical graph with types and attributes on nodes and edges. The main reason we chose Acme is that its modeling environment, AcmeStudio (Schmerl and Garlan 2004; AcmeStudio 2009), is a plugin in the Eclipse tool integration platform, as are many of the other tools that we developed for SCHOLIA.

Most ADLs support the core elements of Acme that SCHOLIA uses : (a) components; (b) connectors; (c) tiers or groups; and (d) hierarchical decomposition to refine a component into a nested sub-architecture (Medvidovic and Taylor 2000).

UML. Runtime architectures have traditionally been of greater interest to academics than to practitioners. The de facto standard for documenting design, UML, added direct support for documenting runtime architectures only recently, with the UML 2.0 standard. With UML 2.0, more UML tools support the manual editing of a runtime architecture. However, existing tools do not yet support extracting a runtime architecture from code, nor do they support analyzing the conformance of an implementation to a target runtime architecture. Overall, the tools for the runtime architecture are still immature compared to the tools available for the code architecture. In particular, analyzing conformance between a runtime architecture and an arbitrary

implementation remains an important but unsolved problem (Shaw and Clements 2006).

1.3.3 Benefits of Architecture

All systems have an architecture, whether it is explicitly documented or not. There are several recognized benefits to documenting the architecture of a system, as I discuss below.

1.3.3.1 System understanding

In object-oriented systems, the dominant pattern-based design methodologies encourage the composition of systems from cooperating objects. So, engineers who want to evolve such an existing system must understand these runtime interactions. In many cases, the architectural documentation may be missing or out of date. When the only reliable source of information is the source code, architects and developers often face the problem of extracting the architecture of the system for the purpose of understanding it.

1.3.3.2 Qualitative architectural evaluation

An architect can document the architecture and use it to qualitatively evaluate risks, tradeoffs and requirements. (Dobrica and Niemel 2002) survey several architectural tradeoff analysis methods. Moreover, sufficient evidence exists about the value of architecture reviews to improve the quality of a system under development (Maranzano et al. 2005). These methods assume that the architecture is known. However, when the architecture is missing or potentially out of date, there must be a way to extract the built architecture from an existing system.

1.3.3.3 Quantitative architectural analysis

Quantitative architectural-level analyses can analyze specific quality attributes such as security (Bidan and Issarny 1997; Moriconi et al. 1997; Deng et al. 2003; Wile 2003; Ren and Taylor 2005; Abi-Antoun et al. 2007b), performance (Spitznagel and Garlan 1998; Williams and Smith 1998) or reliability (Roshandel et al. 2007; Immonen and Niemelä 2008).

These approaches assume that architects have an accurate runtime architecture of the system under study. But in reality, developers often document a system's architecture by hand, and may forget to include all communication that exists in the implementation. Thus, it would be useful to have a principled approach that can extract from an implementation an up-to-date runtime architecture that matches the model required by an architectural-level analysis.

Many architectural analyses rely on assigning architectural properties to the various component and connector instances. For example, an architectural-level security analysis assigns to each component a trustLevel property, which can be either FullTrust, PartialTrust or NoTrust (Abi-Antoun et al. 2006, 2007b). Then, the analysis can check for an *information disclosure* vulnerability, where an attacker steals data while in transit or at rest. For example, this could happen if the trustLevel of the source of a data flow is higher than that of its destination.

Finally, unless the implementation faithfully realizes the carefully designed architecture, the built system may not exhibit the qualities that were carefully thought out. Indeed, the lack of

enforced or checked conformance with the actual implementation remains the Achilles heel of an architecture-based approach (Jackson and Rinard 2000).

1.3.3.4 Avoiding architectural drift and erosion

Missing or un-enforced architectural information is a key factor which contributes to architectural problems, e.g., (Jaktman et al. 1999). These include *architectural drift*, i.e., "a lack of coherence and clarity of form which may lead to architectural violation and increased inadaptability of the architecture" (Perry and Wolf 1992) and *architectural erosion*, i.e., "violations in the architecture that lead to increased system problems and brittleness" (Perry and Wolf 1992). (Hochstein and Lindvall 2005) survey various techniques for combating architectural degeneration, and include, among others, the ability to analyze conformance.

1.4 Architectural Abstraction

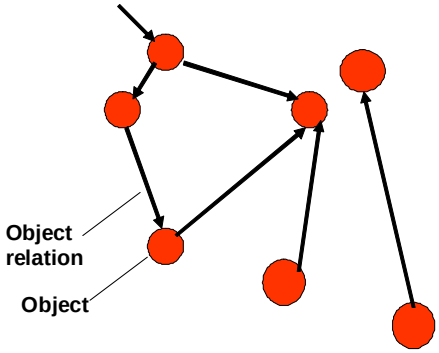
The runtime structure of an object-oriented program can be represented as a *Runtime Object Graph (ROG)*, where nodes correspond to runtime objects, and edges correspond to relations between objects such as points-to field reference relations. It is also possible to show other edges on the object graph, for example, ones that show field accesses or method invocations.

To date, object diagrams were mostly used to show the interactions between a small set of core objects. Because of the immaturity of the tool support for extracting object diagrams from code, many developers have learned to live without them, except perhaps at the design stage.

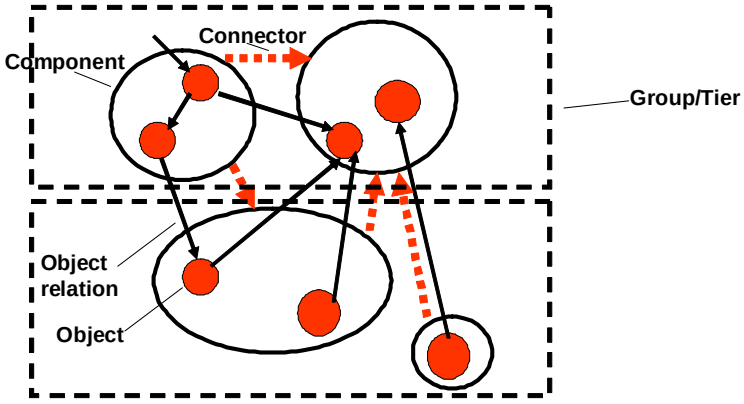
In this dissertation, I argue that object diagrams, once they are hierarchical, scale meaningfully to an entire system, and thus, can also be useful to understand the global application structure of a system. Moreover, such a global object diagram can map fairly intuitively to a runtime architecture of an object-oriented system, which allows reusing much of the existing work in architecture-based approaches.

An object diagram and a runtime architecture are related, but need not be identical. An object diagram and a runtime architecture can differ in the following ways.

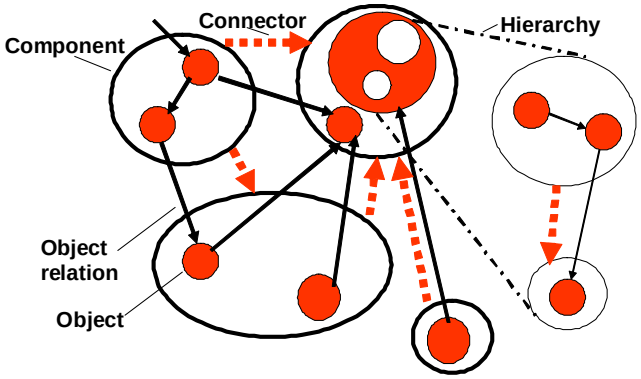
- **An architecture is global:** a runtime architecture is *global*, and shows the object structures for the entire application. On the other hand, an object diagram is often local, and shows the interactions between a few selected objects;
- **An architecture is abstract:** a runtime architecture is potentially more *abstract* than an object diagram. For example, a node in an object diagram typically corresponds to one object or all instances of a given type. But a runtime architecture abstracts one or more objects into conceptual *components*, and represents how those components interact as *connectors* (Clements et al. 2003).
  - **Object abstraction:** a box in an architectural diagram does not necessarily correspond to one object. It could represent multiple instances of the same type, or even different, but related types;
  - **Object clustering:** furthermore, there could be coarser groupings of objects into groups or clusters;



(a) Runtime object graph.



(b) Components, connectors and tiers.



(c) Hierarchical decomposition.

Figure 1.4: Architectural abstraction.

- **Edge abstraction:** an edge in an architectural diagram may correspond to a relation between objects in the implementation. In addition, an edge may correspond to objects in the implementation.

• **An architecture is hierarchical:** a runtime architecture is often *hierarchical*, and can optionally decompose a component into a nested sub-architecture;

Similarly to previous work that defined requirements on architectural description languages

(Luckham and Vera 1995; Shaw and Garlan 1996; Shaw et al. 1995), we define architectural abstraction as follows (Fig. 1.4):

- **Component abstraction.** A runtime architecture shows *components* that correspond to *runtime* entities. For an object-oriented system, a component represents an object or a group of objects. A group of objects must be a meaningful abstraction, for example, from the application domain.
- **Connector abstraction.** An architecture has *connectors* that correspond to relations between runtime entities. For an object-oriented system, a connector represents a runtime interaction between some object in one component and some object in another component.
- **Tier or group abstraction.** An architecture often groups conceptually related component instances into runtime *tiers*, where 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 (Bass et al. 2003). Many architecture description languages (ADLs) have the notion of a tier or *group* (Dashofy et al. 2001).
- **Hierarchical decomposition.** A component can have a nested sub-architecture consisting of lower-level components and connectors. Hierarchy also provides abstraction since it enables both high-level understanding and detail.
- **Scalability.** Large systems would benefit the most from having meaningful, documented architectures. An architecture scales if the size of top-level diagram remains mostly constant as the size of the program increases arbitrarily.
- **Soundness.** Architectural soundness consists of *component soundness*, *connector soundness*, and *tier soundness*:
  - **Component soundness:** An architecture is sound if for every Runtime Object Graph (ROG), there exists a mapping from each runtime object o to exactly one component C in the architecture. In particular, an architecture does not show one runtime entity as two components. Otherwise, an architectural-level analysis may assign these two components different values for a key architectural property, which could invalidate the results of the analysis.
  - **Connector soundness:** If there is a runtime relation between object o_1 and object o_2 in the ROG, then the architecture must have a connector between components C_1 and C_2 corresponding to the communication between o_1 and o_2.
  - **Tier soundness:** If an object o is in a runtime domain d in the Runtime Object Graph (ROG), then the architecture must show component C corresponding to o in the representative D of d.
- **Precision.** An architecture is precise if it shows two runtime entities that represent different conceptual design elements as two different architectural entities. An architecture is imprecise if its elements are too coarse grained and lump together runtime elements that serve different conceptual purposes in the design. For instance, an architecture that represents the entire system as one component is sound, but of course, grossly imprecise. We define precision as:
  - **Component precision:** The architecture shows two runtime entities that represent two different conceptual design elements as two different components.
  - **Connector precision:** The architecture shows two runtime relations that represent two different conceptual interactions as two different connectors.

1.5 Object Graph Extraction

Unfortunately, extracting the runtime architecture of an existing object-oriented system is difficult. In particular, because a system may create many objects at runtime, object diagrams quickly increase in size, even for small systems.

1.5.1 Key Idea: Hierarchical Object Graphs

Hierarchy is often used to mitigate the complexity of a large graph. Hierarchy collapses many nodes into one, and is a classic approach to shrink a large graph. Hierarchy also allows collapsing or expanding selected elements (Storey et al. 2001), to allow both high-level and detailed understanding.

Hierarchy was effective in dynamic object diagrams, e.g., (Hill et al. 2002). Because architectural hierarchy is not readily observable in arbitrary code in a general purpose programming language, imposing hierarchy on a static object diagram is hard. Some language-based solutions, e.g., ArchJava (Aldrich et al. 2002a), extend the language to specify architectural hierarchy and instances directly within the code. But approaches like ArchJava restrict how a program can use objects. As a result, they require re-engineering an existing Java system to follow the more restrictive rules (Aldrich et al. 2002a; Abi-Antoun and Coelho 2005; Abi-Antoun et al. 2007a), a process which is often non-trivial.

1.5.1.1 Annotations to convey architectural intent

To achieve hierarchy in a static object diagram, SCHOLIA combines annotations and static analysis. In SCHOLIA, a developer picks a top-level object as a starting point, then uses local, modular, ownership annotations in the code to impose a conceptual hierarchy on runtime objects. Hierarchy provides architectural abstraction, whereby architecturally significant objects appear near the top of the hierarchy and data structures are further down.

Definition 1 (Abstraction by Ownership Hierarchy and by Types)**.** *A hierarchical object graph provides abstraction by ownership hierarchy when it shows architecturally significant objects near the top of the hierarchy and data structures further down. Moreover, the object graph can provide abstraction by types by collapsing objects further according to their declared types.*

Just as there are multiple architectural views of a system, there is no single right way to annotate a program. Good annotations minimize the number of objects at the top level by pushing low-level objects that are data structures, underneath other, more architecturally significant objects from the application domain. For example, in Aphyds, the annotations make objects of type `Node` or `Net` part of the higher-level `Circuit` object (Fig. 1.5).

In a hierarchical object graph, an object can contain other objects. As a result, many nodes representing lower-level objects can be collapsed underneath a node representing a higher-level object. This is a classic approach to shrink a graph. However, SCHOLIA collapses object nodes based on containment, ownership and type structures, not according to where objects are syntactically declared in the program, a naming convention or a graph clustering algorithm.

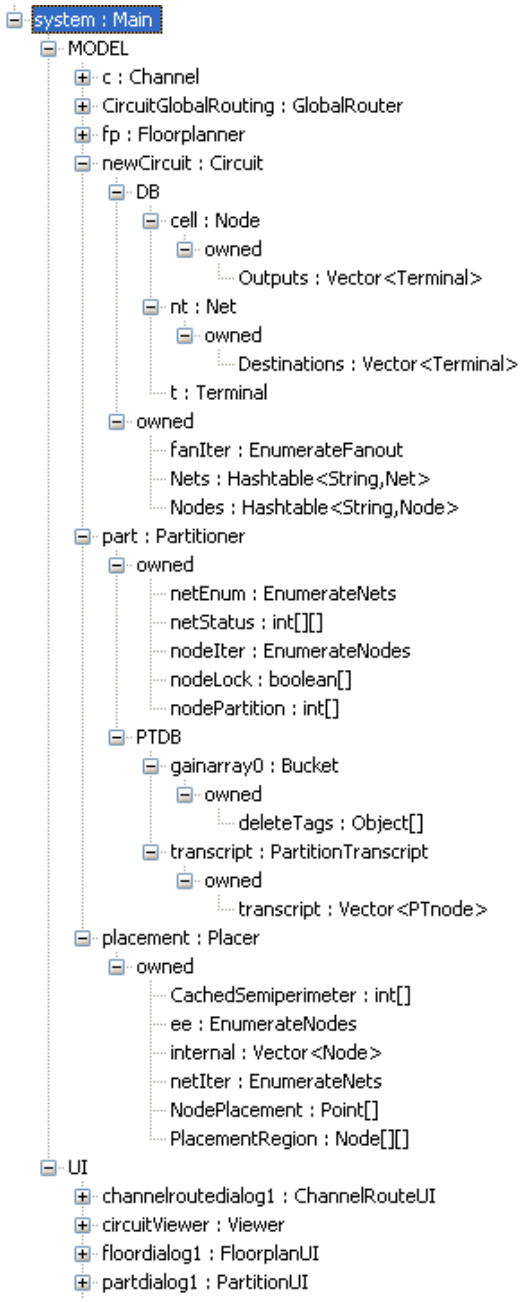


Figure 1.5: Aphyds: partial hierarchy of objects.

1.5.1.2 Static analysis to achieve soundness

A static analysis then extracts from the annotated program a global hierarchical object graph that conveys architectural abstraction by ownership hierarchy and by types. Moreover, the extracted object graph is both *object sound* and *edge sound*.

Definition 2 (Object soundness)**.** *An object graph is object sound if each runtime object has exactly one unique representative in the object graph.*

```
@Domains({"DB"})
class Circuit {
  @Domain("DB") Node node;
  @Domain("DB") Net net;
  ...
}
```

Figure 1.6: Aphyds: `Node` and `Net` objects are *part of* a `Circuit` object.

Definition 3 (Edge soundness)**.** *An object graph is edge sound if it has edges that correspond to all possible runtime points-to relations between the representatives of the runtime objects.*

1.5.2 Example

Instead of placing objects directly inside other objects, SCHOLIA uses an extra level of hierarchy and groups related objects inside a *domain*. A domain is similar to an architectural runtime *tier*, which is a *conceptual partitioning of functionality* (Clements et al. 2003).

This dissertation uses a visualization based on box nesting to indicate the containment of objects inside domains, and that of domains inside objects. For example, the domain DB is inside the object circ (Fig. 1.8). Dashed-border white-filled boxes represent domains. Solid-filled boxes represent objects. Solid edges represent field references. An object labeled `obj:T` indicates an object reference `obj` of type `T`, which we then refer to either as "object `obj`" or as "`T` object", meaning for brevity, "an instance of the `T` class".

SCHOLIA can describe two kinds of hierarchical information, logical containment and strict encapsulation, which I discuss next.

1.5.2.1 Logical containment

The class diagram (Fig. 1.2) shows `Node`, `Net` and `Terminal` classes that are all at the same level as `Circuit`. From the class diagram, it is unclear whether instances of `Node` and `Net` share one `Vector` object.

An architecture often uses hierarchical decomposition to refine a component into a nested sub-architecture (Medvidovic and Taylor 2000). For example, the Aphyds architecture (Fig. 1.1) shows node and net inside circ's substructure.

To define a conceptual group of lower-level objects than an object contains, we use a *public domain*. For instance, a public domain `DB` inside object `circ` contains object `net`. This makes `net` *part of* `circ`. *Part of* means conceptual or logical containment, which we indicate by a thin border. Namely, nested objects may still be accessible to the outside. For instance, any object that can reference `circ` can also reference the child objects `node` and `net` inside the `DB` domain.

A developer indicates this logical containment using annotations (Fig. 1.6). The key idea is to declare a public domain, `DB`, inside `Circuit` and place the `Node` and `Net` objects inside `DB`.

Logical containment can convey arbitrary architectural intent. For instance, the architect could have made a net object conceptually part of the partitioner object, instead of making it part of a circ object (Fig. 1.1). Indeed, the arbitrary nature of architectural intent leaves little hope that a fully automated static analysis could infer meaningful public domains.

```
@Domains({"OWNED"})
class Net {
  private @Domain("OWNED") Vector terms;
  ...
}
```

Figure 1.7: Aphyds: `terms` object is *owned by* a `Net` object.

1.5.2.2 Strict encapsulation

The class diagram (Fig. 1.2) suggests that a `Node` object and a `Net` object might share the same `Vector` object. But at runtime, different instances of `Vector` are often part of conceptually different components. For instance, a `Node` object has a `Vector` object of `Terminal` objects. Another distinct `Vector` object, also of `Terminal` objects, is part of a `Net` object (Fig. 1.8(b)).

Unlike the class diagram which shows one `Vector` class (Fig. 1.2), the runtime structure can distinguish between different instances of `Vector`. Moreover, in this case, we may want to indicate that these `Vector` objects are *strictly encapsulated* or *owned* by other objects. When an object is owned, it is part of another object's private state or *representation*, and no aliases to the owned object can leak to the outside.

A developer indicates that an object is encapsulated by placing it in a *private domain*. For example, `net` has a private domain `OWNED` and object `terms` inside `OWNED` (Fig. 1.7). Our visualization shows a private domain with a thick dashed border (Fig. 1.8(b)). In particular, strict encapsulation guarantees that there can be no incoming references into a `terms` object encapsulated inside a `net` object.

Although `Net` and `Node` objects have their respective distinct `Vector` objects, those two `Vectors` may refer in turn to the same `Terminal` objects that are also in `DB`.

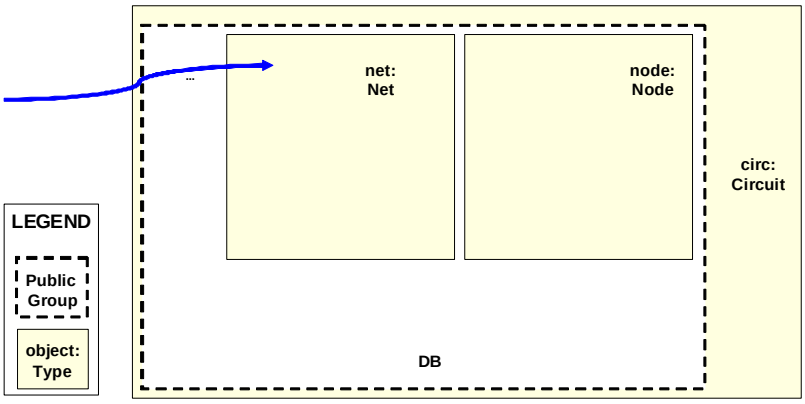
A strictly encapsulated field cannot be assigned to by a public modifier method, or returned from a public accessor method. So there are existing static analyses that can identify strictly encapsulated objects.

1.5.2.3 Sound approximation

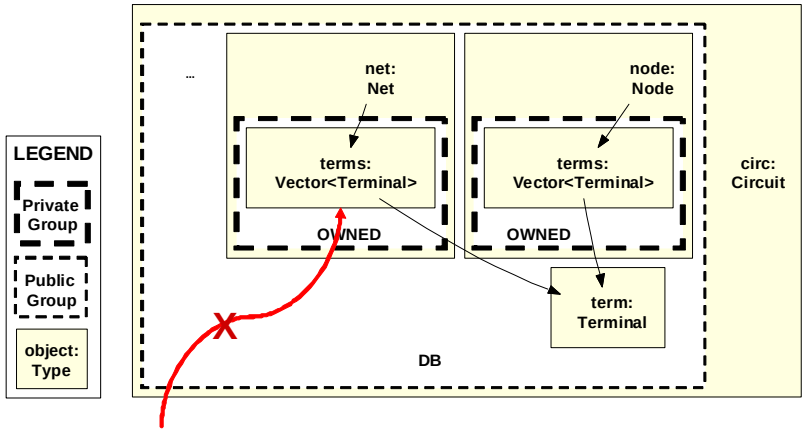
An OOG is an approximation of the actual objects and relations, one that is conservative and may include more objects and relations than those that will actually be there, by virtue of using a sound static analysis. An OOG, like any static object diagram, can be imprecise in several ways (Fig. 1.8(b)). First, it makes no guarantees about the multiplicities of objects at runtime. For example, a given program run of Aphyds may not instantiate a single `Node` or `Net` object. Second, although the diagram shows an edge from `terms` to `term`, a given program run may not actually have such an edge. For instance, the `terms` `Vector` may remain empty during one entire program execution.

1.5.2.4 Aliasing

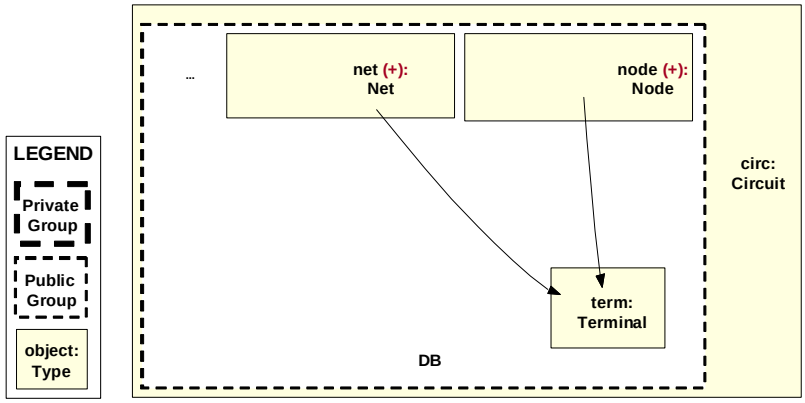
Aliased object must be represented by the same runtime component in the architecture. If an architecture deceptively showed two components for one runtime entity, one could assign these



(a) Representing *logical containment* inside `circ`.



(b) Representing *strict encapsulation* inside `node` and `net`.



(c) Collapsing the sub-structures of `net` and `node`.

Figure 1.8: Aphyds: representing `Circuit`'s runtime sub-structure.

two components different values for a key architectural property such as trustLevel, which could lead an analysis at the architectural level to produce misleading results.

In a program, several object references may alias, i.e., refer to the same object at runtime. Therefore, an alias analysis is needed to identify possible aliasing. In SCHOLIA, we rely instead

on the precision about aliasing that ownership domains offer. In particular, the type system guarantees that two objects in different domains can never alias. But two objects of compatible types, in the same domain, may alias. The analysis uses this information to ensure that the diagram reflects possible aliasing. For instance, consider a field `temps` of type `Stack` that is in the same domain `OWNED` as `Vector` inside `Net`. Since `Stack` is a subtype of `Vector`, the object diagram would display `temps` and `terms` as a single object.

1.5.2.5 Abstraction by hierarchy

SCHOLIA represents a hierarchical object graph as a nested graph with domains and objects inside those domains. Like other such representations, hierarchy allows information at any level to be displayed or elided to show overviews of the system at the desired level of abstraction (Storey et al. 1999).

For instance, Fig. 1.8(c) collapses the substructure of `Node` and `Net` object. A (+) symbol indicates that an object has a collapsed sub-structure. As a result, a low-level object such as `Vector` no longer appears at the same level as `Node` or `Net` objects. Moreover, collapsing the substructure of `Node` and `Net` still represents their relation to `Terminal`. As an aside, note how Fig. 1.8(c) is comparable to the substructure of circuit in the target architecture (Fig. 1.1).

We can use the same nested box visualization to represent the entire Aphyds object tree (Fig. 1.5). Collapsing the sub-structure of most objects produces an object graph that is much more manageable than a flat one. The hierarchical graph (Fig. 1.9) has all the objects that are in the flat graph (Fig. 1.3)[5]. However, the hierarchical graph collapses into one node several objects that are in the flat graph, based on the ownership and the logical containment information of those objects, and optionally, based on their declared types.

In summary, an object-oriented program's runtime structure often bears little resemblance to its code structure. One code element can appear as multiple elements in a runtime structure. In addition, due to possible aliasing, multiple code elements can also correspond to the same element in the runtime structure.

1.5.3 Previous work on architectural extraction

We discuss most of the previous work on architectural extraction in Section 8.6 (Page 291). In summary, previous work in architectural extraction used dynamic analysis, static analysis or a mix of the two. A dynamic analysis takes a snapshot of the heap at runtime, and reveals the structure at that instant in great detail (Flanagan and Freund 2006). Still, it is possible to obtain a high-level picture from the profusion of objects, through the use of extensive graph summarization and manipulation (Mitchell 2006; Mitchell et al. 2009). However, such a snapshot shows one or more executions, meaning it may not reflect important objects or relations that show up only in other executions.

On the other hand, a sound static analysis can extract an object graph that captures all executions. All previous static analyses produce non-hierarchical object graphs that explain runtime

[5]Note, the hierarchical graph shows all the objects that the program may produce except `String` objects, which I purposely excluded. For consistency, I also manually elided `String` objects from the flat graph.

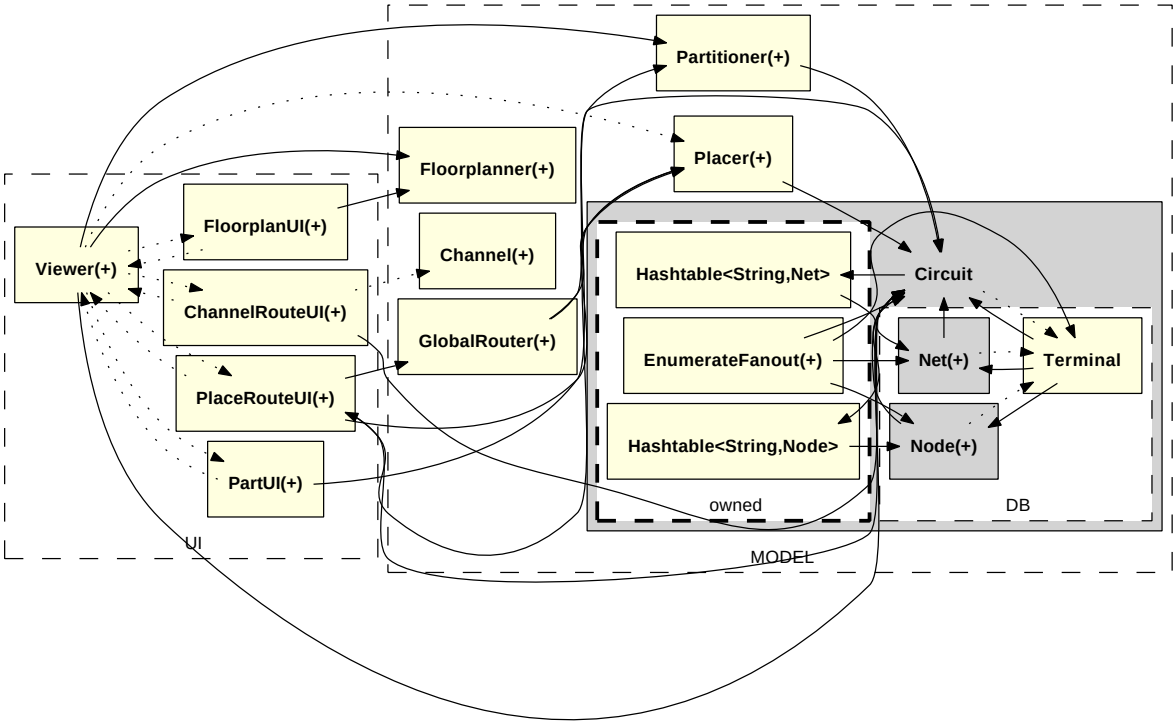


Figure 1.9: Aphyds: hierarchical object graph.

interactions in detail (Jackson and Waingold 2001; O'Callahan 2001; Lam and Rinard 2003), but convey little architectural abstraction, as can be seen in the Aphyds flat object graph (Fig. 1.3).

1.5.4 Summary

SCHOLIA fulfills a previously unexplored space, that of hierarchical static object diagrams. Hierarchy makes an object diagram scale to show the object structures of an entire application, instead of just the interactions between a small set of objects. Furthermore, a hierarchical object graph can provide architectural abstraction and can map intuitively onto a standard runtime architecture.

In Chapter 6, we discuss more precisely how to abstract an object graph into a Component-and-Connector (C&C) view, which is a standard representation of a runtime architecture. Intuitively, a canonical object in an OOG maps to a component, and a domain maps to a tier. Moreover, the abstraction step is largely automated. Even though a developer can control the abstraction, we will almost always use the default options when extracting an OOG and abstracting it into a C&C view, as our evaluation in Chapter 7 will show.

As a result, we will often use the terms "hierarchical object graph" and "runtime architecture" interchangeably. Similarly, we will use the terms "component" and "tier" interchangeably with "object" and "domain", respectively.

1.6 Architectural Conformance

In some domains, it is possible to generate the initial code from an architecture. But developers can still modify the implementation directly and potentially cause it to diverge from the architecture. If the architecture and implementation are inconsistent, the properties that an architect carefully designed into the architecture may not hold in the implementation. Thus, there is value in determining if the implementation conforms to the architecture. Similarly, architects who want to keep their systems evolvable, or maintain various runtime invariants, must ensure that the runtime structure of the built system conforms to the architect's intended architecture. Several researchers have reported that informal architectural diagrams that architects have of their systems, while mostly accurate, often omit important communication that exists in the implementation (Murphy et al. 2001; Aldrich et al. 2002a).

1.6.1 Key Property: Communication Integrity

A system conforms to its architecture if the architecture is a correct abstraction of the runtime behavior of the system. The *communication integrity* property defines one notion of structural conformance, namely how architectural structure constrains runtime communication in the implementation (Moriconi et al. 1995; Luckham and Vera 1995; Aldrich et al. 2002a), as follows:

Definition 4 (Communication integrity)**.** *Each component in the implementation may only communicate directly with the components to which it is connected in the architecture.*

Of course, communication integrity is not the only notion of conformance that may need to be enforced. For example, (Luckham and Vera 1995) identified additional criteria for conformance:

Definition 5 (Decomposition)**.** *For each component in the architecture, there should be a corresponding component in the implementation.*

Definition 6 (Interface Conformance)**.** *Each component in the implementation must conform to its architectural interface.*

In this dissertation, we focus on communication integrity, since it is a fundamental conformance property relating architecture to implementation, upon which several other conformance properties rely (Aldrich 2003). Because communication integrity mandates which components communicate, it provides the foundation for other architectural properties that depend on how these components communicate.

Indeed, many other conformance analyses could be defined. For example, one analysis may enforce a minimum or a maximum in a pool of replicated components. However, there are limits to what can be checked statically because a static object diagram lacks precision on the actual multiplicity of the objects that the program may create, or on the actual relations between objects.

1.6.2 Establishing traceability

When the architecture and the code evolve independently, traceability between the designed architecture and the code is often lost. Once traceability is lost, the development team slowly gives up on having a documented architecture (Jackson and Rinard 2000). Having traceability between the code and a designed runtime architecture has many potential benefits, including

clearer documentation, more focused development, increased system understanding, and a more precise impact analysis of the proposed changes (Lindvall and Sandahl 1996). However, establishing traceability after the fact is still difficult (Spanoudakis and Zisman 2005). The proposed conformance analysis establishes traceability as a side benefit.

1.6.3 Previous work in architectural conformance

We discuss most of the previous work on architectural conformance in Chapter 8.9 (Page 299). In summary, enforcing communication integrity in arbitrary object-oriented implementation code is challenging due to programming language mechanisms that obscure communication pathways, such as references and objects, so previous systems have made serious compromises.

To side-step the problem of architectural conformance, some approaches radically change the programming language to incorporate architectural constructs at the expense of severe implementation restrictions, e.g., (Aldrich et al. 2002a; Schäfer et al. 2008). Others require that developers implement their applications on specialized architectural middleware or frameworks (Medvidovic et al. 1996; Malek et al. 2005), or require an implementation to follow strict style guidelines that prohibit sharing mutable data between components (Luckham and Vera 1995). Still other approaches require developers to always generate parts of the implementation from an architectural model (Shaw et al. 1995; Miller and Mukerji 2003). Finally, to analyze conformance after the fact, previous approaches use dynamic analyses (Sefika et al. 1996b; Schmerl et al. 2006), which, by definition, cannot check all possible system executions.

1.7 The Scholia approach

A general approach to verify the runtime structure must extract a structure that captures all potential executions of a program, then abstract that structure into a high-level representation that is suitable for comparison with the intended architecture.

SCHOLIA enables a developer to extract the built runtime architecture, then use the architecture for documentation, communication, qualitative evaluation or quantitative analysis. For the architectural extraction, SCHOLIA adopts the *extract-abstract-present* strategy (Krikhaar 1997). And for the conformance analysis, SCHOLIA follows the *extract-abstract-check* model (Feijs et al. 1998; Murphy et al. 2001)[6]. The steps are as follows (Fig. 1.10):

1. Add annotations to the code and type-check them;
2. *Extract* a *sound* object graph that conveys architectural abstraction by hierarchy and by types;
3. *Abstract* an extracted object graph into a built runtime architecture;
4. *Present* the built runtime architecture in an architecture description language (ADL) or an architectural modeling environment.

In addition, if the developer can separately document the system's target architecture, he can analyze the conformance of the built architecture to the target, as follows:

1. Document the designed runtime architecture;

[6]Reflexion Models (RM) (Murphy et al. 2001) inspired SCHOLIA heavily, even though RM works only on the code architecture. We compare and contrast the two approaches in more detail in Section 6.6.4 (Page 218).

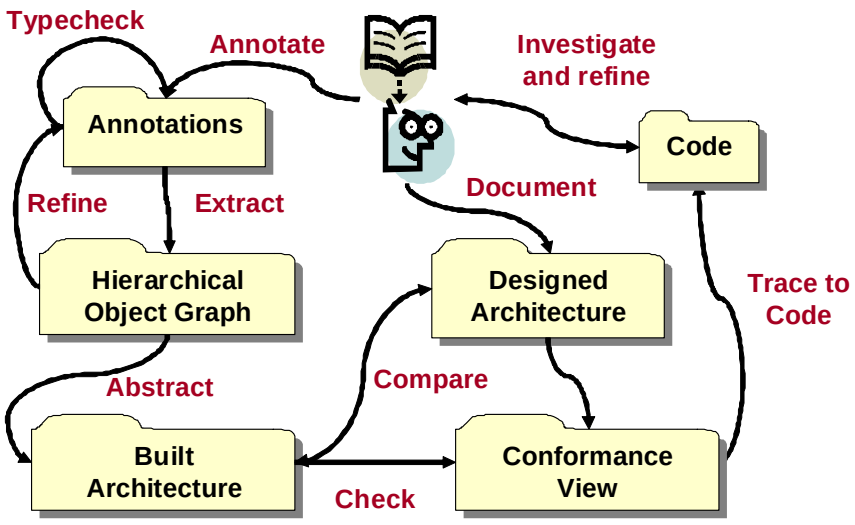


Figure 1.10: Overview of the SCHOLIA approach.

2. Structurally *compare* the built and the designed hierarchical runtime architectures;
3. *Check* their conformance and enforce *communication integrity* in the designed runtime architecture; and
4. Compute a measure of their structural conformance.

Based on the findings, a developer can perform any of the following:

(a) Iteratively refine the annotations based on visualizing an extracted object graph, before abstracting it;
(b) Fine-tune the abstraction of an object graph into an architecture;
(c) Manually guide the comparison of the built and the designed architecture, if the structural comparison fails to perform the proper match;
(d) Correct the code if she decides that the designed architecture is correct, but that the implementation violates the architecture; or
(e) Update the designed architecture if she considers that the conformance analysis highlights an error or omission in the architecture.

Variations. There are several variations on the SCHOLIA approach.

- **Visualize the ownership annotations:** A developer may be interested in adding ownership annotations to detect and prevent aliasing bugs that lead to representation exposure[7]. In that case, SCHOLIA can visualize the ownership structure of an application in order to help a developer fine-tune the annotations. Indeed, one can conceivably use an OOG to judge the quality of the annotations in a program, whether they are added manually or using an inference tool (A. Milanova, personal communication, 2008). Such a judgement is necessarily subjective. A more objective criterion is to compare an OOG to a benchmark, which could be, for example, a target architecture.

[7]The code quality tool FindBugs (`findbugs.sourceforge.net`) uses a shallow, unsound static analysis to warn about possible representation exposure. It is precisely these mistakes that ownership types can prevent.

- **Understand the application's object structures:** Today, a developer can use a number of existing tools to extract a class diagram from the code relatively easily, to help her understand the static code structure of a system. She may want to complement her understanding of the system by studying its runtime structure. So she may be interested in visualizing an object graph and tracing from objects and edges in the object graph to the code.

1.8 SCHOLIA's Requirements

From the above discussion, I list the requirements of a proposed solution. These requirements are mainly based on generally accepted good practices, shortcomings of previous approaches, and the needs for industrial adoptability. In Section 9.1 (Page 305), I return to these requirements and systematically evaluate how SCHOLIA meets each one.

Because SCHOLIA follows the extract-abstract-check strategy, I organize the requirements as being on the overall approach (Section 1.8.1), the annotations (Section 1.8.2), the architectural extraction (Section 1.8.3), the architectural comparison (Section 1.8.4), and the architectural conformance analysis (Section 1.8.5).

1.8.1 Overall Approach

I identify the following requirements on the overall approach:

RQ O1 – Hierarchical architectural models: Modeling a software architecture as a hierarchy of component instances is a generally accepted notion, and many existing ADLs model architecture in this way (Medvidovic and Taylor 2000). Such a model enables a developer to understand the relations between components at a high level, then drill down and study each component recursively.

RQ O2 – Static analysis: Static analysis can extract sound information which considers all possible executions. In contrast, dynamic analysis considers only a few program runs (Sefika et al. 1996b; Schmerl et al. 2006).

RQ O3 – Arbitrary implementation code: To be adoptable, the approach must handle existing object-oriented languages, design idioms and patterns. The approach must also support existing frameworks and libraries, and must not require a specific implementation framework. A developer should not have to re-engineer a system to expose its architecture using an extended language (Aldrich et al. 2002b; Schäfer et al. 2008), or to implement the system on a specialized framework or middleware (Medvidovic et al. 1996; Malek et al. 2005; Bruneton et al. 2006). As (Di Nitto and Rosenblum 1999) pointed out, a middleware often *induces* an architectural style on an application that uses it.

RQ O4 – After the fact analysis: This dissertation focuses on extracting the architecture, and analyzing the conformance of an existing system after the fact. In contrast, model-driven approaches assume that developers always update an architectural model, then generate the code from the architecture to ensure conformance, e.g., (Moriconi et al. 1995; Shaw et al. 1995; Miller and Mukerji 2003). Despite the recent trend in Model-Driven Architecture (Miller and Mukerji 2003), code generation is applicable only in certain domains, as it is often too restrictive, and does not handle legacy code. Whenever developers can directly

modify the implementation, as is often the case, they can potentially introduce architectural violations.

RQ O5 – Automation: The different steps of the approach must be semi- or fully automated.

1.8.2 Annotations

I identify the following requirements for the annotations:

RQ ANN1 – Language support for annotations: The annotations must not extend the language. Instead, they must be structured comments or use available language support for annotations[8]. In addition, the annotations must not affect the program's runtime semantics.

RQ ANN2 – Real object-oriented code: The annotations must support existing object-oriented code that uses aliasing, recursion, inheritance, inner classes, etc.

RQ ANN3 – Expressiveness: The annotations must be expressive and allow annotating a program without having to refactor it to express its architecture. Having to refactor existing code to annotate it increases the cost of adopting the approach.

RQ ANN4 – Automation: A tool must check that the annotations are consistent with each other and with the code. Ideally, a tool also helps with adding the annotations to a program. At least, the annotations should be amenable to automated inference.

1.8.3 Architectural Extraction

The goal of the extraction is to extract an object graph that soundly approximates all possible Runtime Object Graph (ROG)s. I identify the following requirements for the object graph extraction:

RQ EXT1 – Summarization: Different program runs generate a different number of objects. Furthermore, the number of objects in the Runtime Object Graph (ROG) is unbounded. An object graph must be a finite representation of all Runtime Object Graphs (ROGs).

RQ EXT2 – Hierarchy: An object graph must provide architectural abstraction by hierarchy and support both high-level understanding and detail. It must show architecturally significant objects near the top of the hierarchy and data structures further down.

RQ EXT3 – Object soundness: The object graph must show exactly one unique representative for each runtime object.

RQ EXT4 – Edge soundness: The object graph must show edges that correspond to all possible runtime points-to relations between the representatives of the runtime objects.

RQ EXT5 – Traceability: Each node or edge in an object graph should be traceable to a set of nodes from the program's abstract syntax tree, and to the underlying lines of code.

RQ EXT6 – Precision: Ideally, the object graph should have no more edges than soundness requires. However, there may be false positives that are due to infeasible paths. This is an inherent problem in any static object diagram.

RQ EXT7 – Scalability: The static analysis to extract an object graph must scale.

RQ EXT8 – Automation: Tool support must be available to extract an object graph from an annotated program. Furthermore, the extraction tool must have interactive performance.

[8] The C# language supports custom attributes, and Java 1.5 supports annotations (Bloch 2004).

1.8.4 Architectural Comparison

I identify the following requirements for the architectural synchronization:

RQ COMP1 – No unique identifiers: The comparison should not assume that the architectural view elements have unique or persistent identifiers.

RQ COMP2 – No ordering: The comparison should not assume that an architectural view has an inherent ordering among its elements.

RQ COMP3 – Insertions, deletions, and renames: The comparison must handle elements that are inserted, deleted and renamed across two architectural views.

RQ COMP4 – Hierarchical moves: The comparison must detect elements that are moved up or down a number of levels in the hierarchy.

RQ COMP5 – Manual overrides: The user must be able to force or prevent matches between selected view elements. The comparison should then take these constraints into account to improve the overall match.

RQ COMP6 – Type information optional: The comparison should not assume that the view elements have type information that matches exactly. It should be able to recover a correct mapping from structure alone if necessary, or from structure and type information if type information is available. It should, however, take advantage of any available type information, and avoid matching elements that have incompatible types.

RQ COMP7 – Disconnected and stateless operation: The comparison should work after the fact, in a disconnected and stateless mode. In other words, the comparison should not rely on the ability to monitor, intercept, or record the structural changes to an architecture as they occur.

RQ COMP8 – Automation: The comparison must be semi- or fully automated.

1.8.5 Architectural Conformance

I identify the following requirements for the architectural conformance analysis:

RQ CHK1 – Communication integrity: The conformance analysis must enforce *communication integrity*, and must not have false negatives about possible component communication.

RQ CHK2 – Few false positives: Any sound static analysis is bound to generate false positives. However, the rate of false positives must be low. Otherwise, developers will waste most of their time wading through spurious warnings.

RQ CHK3 – Traceability: The conformance analysis should establish traceability between the target architecture and the underlying source files. A developer should be able to trace from each conformance finding to the pertinent lines of code, without having to potentially review the entire code base to investigate a suspected architectural violation.

RQ CHK4 – Automation: The conformance analysis must be fully or semi-automated.

1.9 Contributions

This dissertation contributes SCHOLIA, the first approach to statically extract a hierarchical runtime architecture from existing object-oriented code, requiring only annotations. SCHOLIA is

also the first approach to analyze at compile time communication integrity between code in a widely-used object-oriented language and a rich, hierarchical description of the architect's intended runtime architecture. I break up the overall contribution into the following contributions:

- **Static analysis to extract a hierarchical object graph from a program with ownership annotations.** I designed a novel static analysis to extract a hierarchical object graph, which provides architectural abstraction by ownership hierarchy and by types (Chapter 2). The annotations implement the ownership domain type system (Aldrich and Chambers 2004), and can be checked for consistency with each other and with the code using a tool.
- **Formal validation of soundness.** To validate the object graph extraction algorithm, I represent the core of the algorithm into a formal system incorporating the key constructs of a Java-like language and prove soundness properties (Chapter 3).
- **Evaluation of the annotations and the object graph static analysis.** I improved the tool support for the ownership domain type system, then used the tools to add annotations to real object-oriented code. To my knowledge, these are some of the largest and most substantial case studies in evaluating ownership types. In addition, I implemented the static analysis to extract object graphs, and extracted meaningful hierarchical object graphs from several representative systems that I annotated manually (Chapter 4).
- **Novel comparison of hierarchical architectural views.** I developed a novel approach for structurally comparing two hierarchical architectural views (Chapter 5). Using structural information enables detecting elements that are inserted, deleted, renamed, or moved up or down in a hierarchy. In contrast, previous approaches to differencing architectural views assume that view elements have unique node identifiers, which is often not the case. Other approaches detect only insertions and deletions, and as a result, lose the properties of architectural elements, upon which several architectural-level analyses rely.
- **Novel techniques to abstract an object graph into a built runtime architecture, then analyze conformance between a built and a target architecture.** An extracted object graph may not be isomorphic to the architect's intended architecture, making further abstraction necessary. I specialized the view synchronization approach, which makes two views identical, to analyze conformance. The conformance analysis allows a designed architecture to be more abstract than a built architecture. Still, SCHOLIA soundly summarizes in the designed architecture any additional communication that is present in the implementation, without propagating low-level implementation objects into the designed architecture. For example, SCHOLIA can represent some objects in the built architecture as part of a connector in the designed architecture (Chapter 6).
- **Evaluation of the end-to-end conformance analysis approach.** Using case studies, I demonstrate that, in practice, SCHOLIA can be applied to existing systems while changing only annotations in the code, that SCHOLIA can find interesting architectural violations, that these violations can be traced to code, and that SCHOLIA computes sensible conformance metrics (Chapter 7).

1.10 Thesis Statement and Outline

The thesis of this dissertation is:

SCHOLIA *can extract a sound, hierarchical, runtime architecture from an existing object-oriented system and analyze communication integrity with a target architecture, entirely statically and using typecheckable ownership annotations.*

I created several corresponding hypotheses, subordinate to the main thesis. Since each hypothesis is smaller than the main thesis, each can be directly supported by evidence. Taken together, these hypotheses solve the problem of architectural extraction and conformance analysis, for an important class of object-oriented systems.

1.10.1 Hypothesis: Annotations

H-1: Lightweight typecheckable ownership annotations can specify, within the code, local hints about object encapsulation, logical containment and architectural tiers.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- Ownership domain annotations are a natural expression of architectural intent in practice, i.e., they capture software engineering intuition;
- It is possible to annotate existing object-oriented code that uses the Java standard library or other third-party libraries;
- It is possible to use existing language support for annotations, software development tools and integrated development environments, without requiring language extensions;
- An annotated program has few remaining annotation warnings;
- Successfully annotating an existing program requires no or few changes to the code;
- By adding annotations, a developer can detect code-level violations of the architectural intent.

Evidence. We support this hypothesis with the following evidence:

1. I evaluate the annotations on several representative, extended examples of medium-sized Java programs, developed by others, using the success criteria above (Chapter 4).
2. The evaluation shows that, in practice, a developer can capture as program annotations some of his architectural intent. Some of that intent may be currently captured as informal comments in the code or informal architectural diagrams.
3. We present concrete examples of how, in practice, the annotations can effectively help a programmer identify design problems such as tightly coupled code and suggest ways to refactor the code, e.g., by programming to an interface or using a mediator.

1.10.2 Hypothesis: Extraction

H-2: In practice, a static analysis can extract from an annotated program a global, hierarchical object graph that provides architectural abstraction by ownership hierarchy and by types.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- An extracted object graph has fewer objects at the top level, compared to a flat object graph, due to the effective abstraction of objects by ownership hierarchy and by types;
- An extracted object graph does not show low-level objects that are data structures at the top level;
- An extracted object graph rarely suffers from too much or too little abstraction that lead to a useless representation. E.g., rarely does an extracted object graph appear as a fully connected graph, or show one box for the entire system;
- The hierarchy in an extracted object graph corresponds to the system decomposition in architectural diagrams;
- An extracted object graph can help a developer improve the quality of the annotations by encouraging her to push more objects underneath other objects to reduce clutter at the top level;
- An extracted object graph provides overviews of a system's runtime structure at various levels of abstraction;
- An extracted object graph can give insights into the system's runtime structure by identifying undocumented information, contradicting documented information or highlighting interesting structural information.

Evidence. We support this hypothesis with the following evidence:

1. A definition of a static analysis to extract a global object graph from a program with ownership domain annotations (Chapter 2);
2. An evaluation of the static analysis on several real object-oriented systems (Chapter 4), using the success criteria above;
3. A detailed description of the different choices a developer can make to extract a meaningful object graph from an annotated program.

1.10.3 Hypothesis: Soundness

H-3: Each extracted object graph is sound, i.e., it maps each runtime object to exactly one node in the object graph, and represents all edges between runtime objects, in any program run.

Evidence. We support this hypothesis with the following evidence:

1. A formal definition of the core of the analysis using abstract interpretation (Chapter 3);
2. A formal proof of *object soundness* and *edge soundness* (Chapter 3).

1.10.4 Hypothesis: Abstraction

H-4: An analysis can abstract an object graph into a component-and-connector runtime architecture in a standard architecture description language.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- A developer can apply the abstraction techniques, without having to manually select and elide individual objects or domains.

Evidence. We support this hypothesis with the following evidence:

1. A definition of a mapping between a hierarchical object graph and a standard architecture description language (Chapter 6);
2. An evaluation of the approach on several real object-oriented systems (Chapter 7).

1.10.5 Hypothesis: Comparison

H-5: An analysis can structurally compare the built architecture to a documented target runtime architecture.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- A developer can use the comparison, without having to manually force or prevent matches between the majority of individual objects or domains.

Evidence. We support this hypothesis with the following evidence:

1. A definition of an approach for differencing and merging hierarchical architectural views based on structural information (Chapter 5);
2. An evaluation of the approach on several real runtime architectures for object-oriented systems (Chapter 5).

1.10.6 Hypothesis: Conformance

H-6: An analysis can analyze communication integrity against a target architecture, establish traceability between the target architecture and the code, and compute structural conformance metrics in practice.

Success criteria. The success criteria to objectively measure or falsify this hypothesis include:

- The approach can show the absence or presence of a relation or communication between two components, one that was previously unknown, and possibly a sign of bad coupling;
- The approach can provide positive assurance that the code conforms to an intended architecture;
- The approach can help a developer find and reconcile interesting differences between an implementation and a target architecture. A finding is interesting if it identifies undocumented information, contradicts available documentation, or highlights a potential design or implementation defect.
- A developer can investigate a suspected code-level violation of the conformance policy by tracing from the extracted architecture to the relevant lines of code without having to potentially review the entire code base, thus making the warning actionable;

- A tool can enforce structural constraints on the extracted architecture using architectural constraints, types and styles. A subject system could follow or nearly follow some of these constraints. Of course, the structural constraint must have some rationale, e.g., to satisfy quality attributes such as security or performance. For example, if the architecture dictates a pipeline according to the Pipe-and-Filter style, where components are connected in sequence, the tool raises a warning if the built architecture shows connections that bypass elements of the sequence or form a cycle.

Evidence. We support this hypothesis with the following evidence:

1. An end-to-end approach for enforcing communication integrity in a target architecture (Chapter 6);
2. An evaluation of the approach on several real object-oriented systems (Chapter 7), using the success criteria above.

1.11 Summary

The quote at the beginning of the chapter from the landmark Design Patterns book emphasizes the need for understanding a system's runtime architecture, together with its code architecture (Gamma et al. 1994). This dissertation proposes SCHOLIA, a principled approach to extract the runtime architecture of an arbitrary system written in a general purpose programming language, using annotations. Moreover, if a target architecture exists, SCHOLIA can analyze its conformance with the implementation, and enforce communication integrity in the target architecture.

Such an approach can increase the effectiveness of reasoning architecturally about existing systems, because it ensures that the architecture is a faithful representation of the code, which is ultimately the most reliable and accurate description of the built system.

Chapter 2

Object Graph Extraction[1]

In this chapter, I describe informally how SCHOLIA uses annotations and a static analysis to extract a hierarchical object graph that provides architectural abstraction by ownership hierarchy and by types.

2.1 Introduction

A *Runtime Object Graph (ROG)* represents the runtime structure of an object-oriented program. Nodes correspond to runtime objects. Edges correspond to relations between objects such as points-to field reference relations. The goal of the object graph extraction static analysis is to construct a hierarchical object graph that soundly approximates any ROG that any program run may generate.

The rest of this chapter is organized as follows. In Section 2.2, I illustrate the differences between the code and the runtime structure using Listeners, a system smaller than Aphyds (Chapter 1.2.1, Page 2). Section 2.3 presents the annotations that specify architectural intent in the code. Section 2.4 presents a static analysis that extracts an object graph by abstract interpretation over the annotated program. I discuss various advanced features in Section 2.5 and conclude with a discussion in Section 2.6.

2.2 Code vs. Runtime Structure

In this chapter, I use as a running example the Listeners system, a small Document-View architecture. In Listeners, `BarChart` and `PieChart` objects render a `Model` object. All classes implement a `Listener` interface. I chose this example because empirical data shows that listeners are often hard to understand in object-oriented code (Lee et al. 2008, Table 2)[2].

For presentation purposes, I simplified the Listeners example (the code is in Fig. 2.1). In particular, the `Listener` interface does not have a `notify()` method, that all the classes imple-

[1] Portions of this chapter appeared in (Abi-Antoun and Aldrich 2007b, 2008b, 2009a).

[2] (Lee et al. 2008) report the following quote from a participant in an exploratory user study: "If you have [many] system listeners, where people register methods or classes to callback [. . . an] interesting visualization would be [. . .] to explore the actual instances of classes at run-time; it would be better than the list of listeners".

```
1  interface Listener {
2  }
3
4  class BaseChart implements Listener {
5    private List<Listener> listeners = new List<Listener>();
6  }
7
8  class BarChart extends BaseChart {
9  }
10
11 class PieChart extends BaseChart {
12 }
13
14 class Model implements Listener {
15    private List<Listener> listeners = new List<Listener>();
16 }
17
18 class Main {
19   Model model = new Model();
20   BarChart barChart = new BarChart();
21   PieChart pieChart = new PieChart();
22 }
```

Figure 2.1: Listeners: code without annotations.

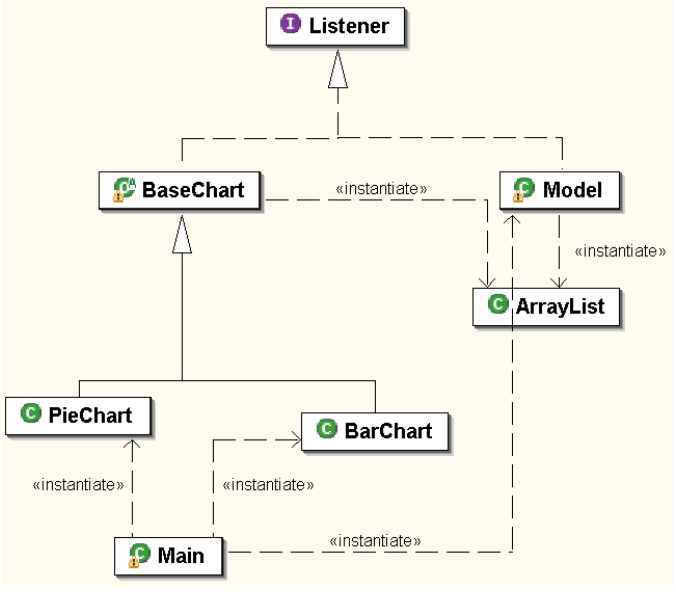
menting the interface have to implement. Moreover, I treat `List` as a class, although in the Java Standard library, `List` is an interface that is implemented by concrete classes such as `ArrayList`. Also, in the following discussion, when I refer to "a `BarChart` object", I mean "an instance of the `BarChart` class".

2.2.1 Code Structure

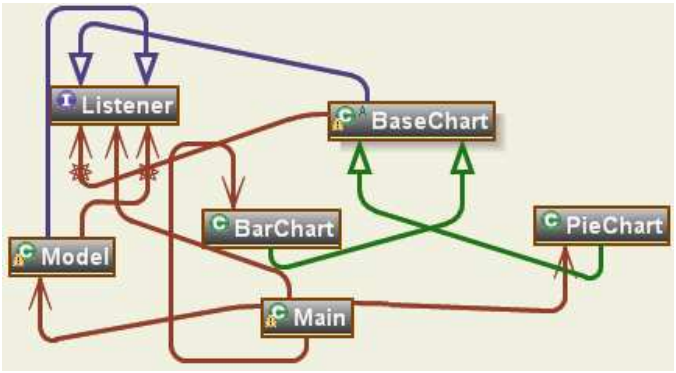
A developer evolving an object-oriented system needs to understand the type structure of the program, which is typically represented as a class diagram. Today, many tools can extract such class diagrams from code. For example, I used EclipseUML (Omondo 2006) and AgileJ (AgileJ 2008) to extract class diagrams from the Listeners program (Fig. 2.2).

Fig. 2.2 shows classes, inheritance and association relations. For instance, classes `BarChart` and `PieChart` extend from `BaseChart`. `BaseChart` and `Model` implement a `Listener` interface. The diagram also shows associations from `Model` and `BaseChart` to `List`. A class diagram explains the type structure of an application but sheds little light on its runtime structure. From the class diagram, it is unclear whether instances of `PieChart` and `BarChart`, which inherit from `BaseChart`, share one `Listener` object.

In a class diagram, it is also common to see several classes depend on a single container class such as `List` or `Vector`. However, different instantiations of such a class often have distinct conceptual purposes and correspond to different elements in the design. Based on the class diagram, it is unclear if instances of `PieChart` and `BarChart` share one `List` object. For instance, a reference of type `Listener` inside an object of type `List<Listener>` can correspond to multiple



(a) Code architecture extracted by Eclipse UML.



(b) Code architecture extracted by AgileJ.

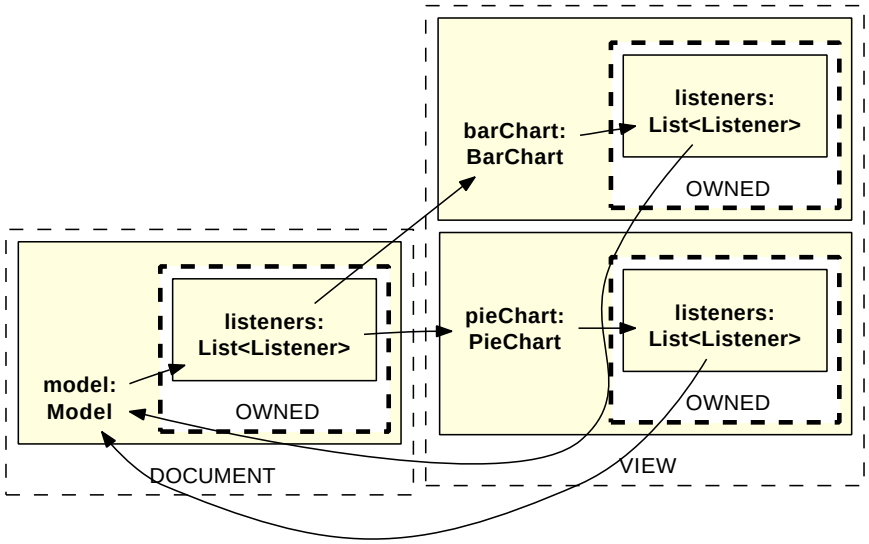
Figure 2.2: Listeners: class diagrams.

design elements, based on the context. Inside the `Model` class, a list element of type `Listener` refers to an object of type `BaseChart` or one of its subclasses. But inside the `BaseChart` class, a list element of type `Listener` refers to an object of type `Model`.

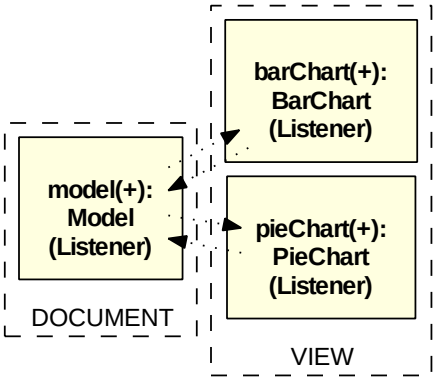
2.2.2 Runtime Structure

A developer also needs to understand the runtime structure of an application, which is often represented as an object diagram. Unfortunately, the tools to extract meaningful object graphs from arbitrary object-oriented code are less mature than the tools that extract class diagrams.

Fig. 2.3(a) shows the runtime structure of the application, and uses the following graphical conventions. Box nesting indicates hierarchical containment. Dashed white-filled boxes represent conceptual groups of objects or tiers. A solid border grey-filled rectangle with a bold label



(a) Hierarchical runtime architecture.



(b) Overview architecture.

Figure 2.3: Listeners: hierarchical object graphs.

represents an object. A solid edge represents a field reference between two objects. An object labeled "obj : T" indicates an object of type T as in UML object diagrams. For example, `barChart:BarChart` indicates a `barChart` reference that is of type `BarChart`.

Conceptually, each view has a separate `listeners` collection object, and the `listeners` object of a `pieChart` is distinct from that of a `barChart` (Fig. 2.3(a)). In a runtime view, we model these lists as *part of* a `barChart` or `model`. At runtime, `BarChart` and `Model` objects each contain a `List` of `Listener` objects.

An analysis for object-oriented code must handle inheritance. In this case, `PieChart` and `BarChart` extend a super class, `BaseChart`, and it is `BaseChart` that declare the `listeners` field. In addition, there is possible aliasing. If the `listeners` field of `BarChart` and `Model` referred to the same object at runtime, the architecture in Fig. 2.3(a) would be deceptive; a correct architecture must show them as one object.

In many object-oriented design patterns, much of the functionality is determined by what in-

stances point to what other instances. For instance, in the Observer design pattern (Gamma et al. 1994, p. 293), understanding "what" gets notified during a change notification is crucial for understanding the functioning of the system, but "what" does not usually mean a class, "what" means a particular instance.

For instance, Fig. 2.3(a) highlights that a `model` object is potentially registered as a listener for a `barChart` object, but a `pieChart` object and a `barChart` object are not registered as listeners to each other.

Ideally, an architecture "can be read in 30 seconds, in 3 minutes, and in 30 minutes" (Koning et al. 2002). In Fig. 2.3(b), we elided the sub-structures of `barChart`, `pieChart` and `model`, and no longer show the various `List` objects (the (+) symbols on the object labels remind us of the elided substructures). In addition, the dotted edges summarize any solid edges by lifting them from elided objects to visible ones.

2.3 Annotations

SCHOLIA's principled architectural extraction combines type annotations and a static analysis. A developer guides the architectural abstraction by adding annotations to the source code to clarify the architectural intent. Because architectural hierarchy is not readily observable in arbitrary code, the annotations specify, within the code, object encapsulation, logical containment and architectural tiers, which are not explicit constructs in general purpose programming languages.

2.3.1 Object and Domain Annotations

The SCHOLIA annotations implement the ownership domain type system (Aldrich and Chambers 2004), which I review while explaining the annotations that a developer might add to the implementation of the Listeners system (Fig. 2.4).

Definitions. An *ownership domain* is a conceptual group of objects with an explicit name and explicit policies that govern how it can reference objects in other domains (Aldrich and Chambers 2004). The annotations assign each object to a single ownership domain that does not change at runtime. A developer indicates what domain each object is part of by annotating each reference to that object in the program. A typechecker validates the annotations and identifies where the annotations are inconsistent with each other or with the code.

The annotations also describe policies, called *domain links*, that govern object references between ownership domains (we explain domain links and give examples in Section 2.3.2).

Graphically, our visualization uses a white-filled rectangle with a dashed border to represent an ownership domain. We also label each rectangle with the domain name.

Annotation syntax. This dissertation often uses a simplified annotation syntax that extends the language (Fig. 2.5). The syntax is similar to the one used by the formal system (Fig. 3.1, Page 77) with one difference. The annotation syntax emphasizes the semantic difference between the owner domain of an object and its domain parameters, whereas the formal system treats the first domain parameter of a class as its owning domain.

```
interface Listener {
}
class BaseChart<M> // Declare domain parameter M
      implements Listener {
  domain OWNED; // Declare protected domain OWNED
  // Outer OWNED annotation is for the list object
  // List has domain parameter ELTS for its elements
  // Nested inner M annotation is bound to List's ELTS for the list elements
  OWNED List<M Listener> listeners = new List<M Listener>();

  // A public method CANNOT return a reference to an object in a private domain
  // So the following lines of code are commented out on purpose
  // public OWNED List<Listener> getListeners() {
  // return listeners;
  //}
}
class BarChart<M> extends BaseChart<M>  {
}
class PieChart<M> extends BaseChart<M>  {
}
class Model<V> implements Listener  {
   domain OWNED;
   // Inner annotation V is for the list elements
   OWNED List<V Listener> listeners = new List<V Listener>();
}
class Main {
  domain DOCUMENT, VIEW; // Top-level domains
  // Bind domain parameter V to actual domain VIEW
  DOCUMENT Model<VIEW> model = new Model<VIEW>();
  VIEW BarChart<DOCUMENT> barChart = new BarChart<DOCUMENT>();
  VIEW PieChart<DOCUMENT> pieChart = new PieChart<DOCUMENT>();
}
```

Figure 2.4: Listeners: code with annotations.

Concrete annotation language. The concrete annotation system and tools use existing language support for annotations, which tends to be verbose (Fig. 2.6). Appendix A has more details on the concrete annotation language as well as examples in that language.

Code examples. In addition, we simplified the code snippets included in this document to show only class and field declarations with their annotations, and ignore Java language features such as methods, generic types, and casts.

Domain names. A developer typically chooses domain names that convey some architectural intent, such as `DOCUMENT` or `VIEW`. In this document, I often show domain names in capital letters to distinguish them from other program identifiers, since most coding conventions discourage the use of all capital letters for non-constants.

$$
\begin{array}{rcl}
P \in Program & ::= & (\overline{L}, C, e) \\
L \in ClassDecl & ::= & \texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}\ [\texttt{extends}\ C'\texttt{<}\overline{\beta}\texttt{>}] \\
 & & \{\ \overline{D};\ \ \overline{F};\ \ \overline{M}\ \} \\
D \in DomDecl & ::= & [\texttt{public}]\ \texttt{domain}\ d; \\
F \in FieldDecl & ::= & T\ f; \\
M \in MethDecl & ::= & \ldots \\
n & ::= & d \mid v \\
p & ::= & \alpha \mid n.d \mid \texttt{shared} \\
T \in Type & ::= & p_{owner}\ C\texttt{<}\overline{p_{params}}\texttt{>} \\
\alpha, \beta \in DomParam & & C, C' \in ClassName
\end{array}
$$

Figure 2.5: Simplified annotation syntax. Adapted from the formal system (Fig. 3.1, Page 77). We excluded domain links for simplicity.

```
1  @Domains({"OWNED"})
2  @DomainParams({"M"})
3  abstract class BaseChart implements Listener {
4    @Domain("OWNED<M>") List<Listener> listeners = new List<Listener>();
5  }
6  @Domains({"OWNED"})
7  @DomainParams({"M"})
8  @DomainInherits({"BaseChart<M>"})
9  class BarChart extends BaseChart {
10 }
11 ...
12 @Domains({"DOCUMENT","VIEW"})
13 class Main {
14   @Domain("DOCUMENT<VIEW>") Model model = new Model();
15   @Domain("VIEW<DOCUMENT>") BarChart barChart = new BarChart();
16   ...
17   public static void main(@Domain("lent[shared]")String[] args) {
18     @Domain("lent") Main system = new Main();
19   }
20 }
```

Figure 2.6: Listeners: code with the concrete annotations.

Declaring a domain. Each class can declare one or more domains to hold the objects that make its parts, thus supporting hierarchy. A domain can be *private* or *public* to distinguish between private or externally-visible state.

Private domains. A private domain, such as `OWNED` (line 5 in Fig. 2.4), provides *strict encapsulation*. For instance, a `public` method cannot return an alias to an object inside a private domain, even though the Java type system allows returning an alias to a field marked as `private`. Thus, instance encapsulation is stronger than making a field be `private` to restrict its module visibility. For example, the `listeners` collection object inside `barChart` is encapsulated. The

typechecker will produce a warning if I were to define a public method inside class `BarChart` that returns an alias to `listeners`. A correct implementation of such a method, however, could return a shallow copy of the `List` object, to avoid the representation exposure.

Public domains. A public domain provides *logical encapsulation*. Having access to an object gives the ability to access all the objects inside its public domains. For example, instead of encapsulating the `listeners` object by placing it inside the private domain `OWNED`, I could define a `LISTENERS` public domain and place `listeners` inside `LISTENERS`. Then, any object that has access to a `barChart` object gets the ability to access the `listeners` instance. I present these alternate annotations in Section 2.6.

Distinguishing between private and public domains. Graphically, our visualization distinguishes between private and public domains, by showing a private domain with a thick dashed border, and a public domain with a thin dashed border.

Top-level domains. SCHOLIA assumes that the program operates by creating a main object. I refer to the domains declared by the class of the root object as the *top-level domains*.

Domain parameters. Domain parameters allow objects to share state and work as follows. An object *X* can access objects in a domain *D* of object *Y* by declaring a formal *domain parameter* on the class of *X* and *binding* that formal domain parameter to domain *D* as long as *domain link* permissions allow *X* to access *D* (we discuss domain links further in Section 2.3.2). Wherever the program instantiates a class that declares domain parameters, the domain parameters must be bound to other domains that are in scope. Note, the class of the root object declares no domain parameters. Graphically, our visualization represents a formal domain parameter with a white-filled rectangle with a dotted border.

For example, class `BarChart` needs to access objects in the `DOCUMENT` domain that is declared in class `Main`. So `BarChart` declares a domain parameter `M` (line 3). When class `Main` declares an object of type `BarChart`, it binds `BarChart`'s domain parameter `M` to its locally declared domain, `DOCUMENT` (line 30), so that a `BarChart` instance can refer to other objects inside `DOCUMENT` such as `model`.

Domain parameters must also be bound to account for inheritance. For example, `BaseChart` takes a domain parameter `M`. So each subclass of `BaseChart`, such as `BarChart` and `PieChart`, binds its domain parameter `M` to `BaseChart`'s `M` domain parameter (lines 17, 19).

Why domain parameters? I glossed over why `BaseChart`, `BarChart`, `PieChart` and `Model` required domain parameters (Fig. 2.4). They do, because they all use `List` which is part of the Java standard library (Fig. 2.7). Recall, here we use `List` as if it were a a concrete class such as `ArrayList`.

Library code is often parametric with respect to application components. For example, the `List` class is parametric in two ways. First, `List` is parametric in the type of the element stored

```
1  // T is generic type parameter
2  // ELTS is a domain parameter for the list elements
3  class List<ELTS T> {
4    private domain OWNED; // Private domain
5    // Place the list's representation in a private domain
6    OWNED Object[] rep;
7
8    // A list has virtual references to the elements it holds.
9    // A virtual field declaration can simulate that.
10   ELTS T obj;
11 }
```

Figure 2.7: Class `List` is parametric in the ownership domain of its elements.

in the list, hence the `T` type parameter. `List` also takes a formal domain parameter, `ELTS` (line 3), that specifies the domain of the elements stored in the list[3].

Back in the Listeners example (Fig. 2.4), the outer `OWNED` annotation, inside class `BarChart`, is for the `List` instance itself (line 9). The inner `M` annotation binds the formal domain parameter `ELTS` to `BarChart`'s domain parameter `M` (line 9), to allow the `List` to access objects inside `M`.

Why ownership domains? SCHOLIA adopts ownership domains because of the expressiveness of the type system, and its suitability for representing architectural intent in code. In principle, SCHOLIA could use an ownership type system that assumes a single *context* per object (Clarke et al. 1998). However, having multiple domains per object is often useful for modeling architectural runtime tiers.

In addition, ownership domains have a crucial expressiveness advantage that can reduce the number of objects in the top-level domains in an extracted architecture. In an owner-as-dominator type system, any access to a child object must go through its owning object (Clarke et al. 1998). In contrast, the ownership domain type system supports pushing almost[4] any object underneath any other object in the ownership hierarchy. A child object may or may not be encapsulated by its parent object: a child object can still be referenced from outside its owner if it is part of a public domain of its parent, or if a domain parameter is linked to a private domain (Aldrich and Chambers 2004).

If making an object owned by another object restricts access to the owned object, then adding annotations to existing code, after the fact, would force more objects to be peers, and thus lead to more cluttered object graphs. On the other hand, using *logical containment* with public domains is more flexible than the *strict encapsulation* of private domains, and can also reduce the number of objects in the top-level domains.

Owner-as-dominator. Still, ownership domains can also enforce the strict owner-as-dominator discipline found in other ownership type systems. To fully encapsulate an object, a developer can declare an object reference in a domain that satisfies the following conditions: (a) the domain

[3]Typically, we annotate the `List` class to take a single domain parameter to store the list's elements, which means that all the objects referenced by a `List` object are in the same domain.

[4]A well-formed ownership relation cannot have cycles.

```
1  class Sequence<ELTS> assumes OWNER -> ELTS {
2    domain OWNED; // Private domain
3    public domain ITERS;
4    link OWNED -> ELTS;
5    link ITERS -> ELTS, ITERS -> OWNED;
6
7    private OWNED Cons<ELTS> head;
8
9    public void add(ELTS Object o) {
10     head = new Cons<ELTS>(o,head);
11   }
12
13   public ITERS Iterator<ELTS> getIter() {
14     return new SequenceIterator<ELTS, OWNED>(head);
15   }
16 }
17
18 class Cons<ELTS> assumes OWNER -> ELTS {
19   ELTS Object obj;
20   OWNER Cons<ELTS> next;
21
22   Cons(ELTS Object obj, OWNER Cons<ELTS> next) {
23     this.obj=obj; this.next=next;
24   }
25 }
```

Figure 2.8: `Sequence` abstract data type with ownership domains.

is `private`; and (b) there is no domain link from any of the formal domain parameters of the declaring class to the `private` domain (Aldrich and Chambers 2004). Placing an object o inside such a domain fully encapsulates o.

2.3.2 Permission Annotations

Objects within a single ownership domain can refer to one another, but references can only cross domains if the programmer specifies a *domain link* between the two domains when they are created (Aldrich and Chambers 2004). A domain link is a policy that an object can declare to describe the permitted aliasing among objects in its internal domains, and between its internal domains and external domains. Ownership domains support two kinds of policy specifications:

- A domain link from one domain to another, denoted with a dashed arrow in the diagram, allows objects in the first domain to access objects in the second domain;
- A domain can be declared public. Permission to access an object automatically implies permission to access its public domains.

For example, `Sequence` uses a linked list as its internal representation. So it places those `Cons` objects in the private `OWNED` domain (Fig. 2.8). `Sequence` also defines a public domain, `ITERS`, to holds the iterator objects. A domain link from the `ITERS` domain to the `OWNED` domain allows those iterator objects to access the list's representation in the `OWNED` domain. Both domains `ITERS` and `OWNED` can access the domain parameter `ELTS`. The `ITERS` domain is public,

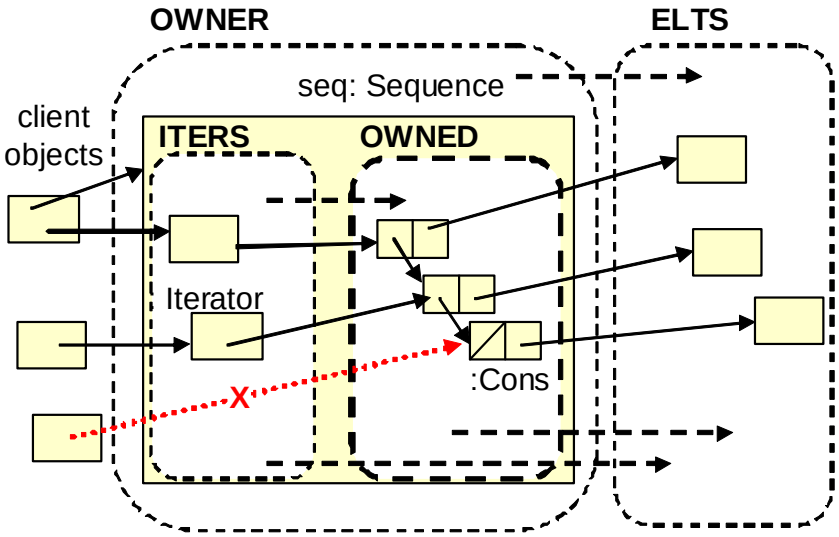


Figure 2.9: A conceptual view of the `Sequence` abstract data type. Dashed edges represent link permissions between domains.

allowing clients to access the iterators. But the `OWNED` domain is private, so outside objects cannot directly access the `Cons` objects. Instead, the clients must access the elements of a `Sequence` object through its iterator interface rather than traversing the linked list directly. A graphical representation of the domains and the domain links is in Fig. 2.9. Graphically, our visualization represents a domain link between two ownership domains with a dashed edge.

In addition to the explicit policy specifications mentioned above, the following policy specifications are implicit:

1. An object has permission to access other objects in the same domain;
2. An object has permission to access objects in all of the domains that it declares.

The first rule allows the different `Cons` objects in the linked list to access each other, while the second rule allows the sequence to access its iterators and linked list. Any reference not explicitly permitted by one of these rules is prohibited, according to the principle of least privilege. It is crucial that there is no transitive access rule. For example, even though clients can refer to iterators and iterators can refer to the linked list, clients cannot access the linked list directly because the sequence has not given them permission to access the `OWNED` domain. Thus, the policy specifications allow developers to specify that some objects are an internal part of an abstract data type's representation, and the typechecker enforces the policy, ensuring that this representation is not exposed.

2.3.3 Special Annotations

Several special annotations add expressiveness to the type system, and can be considered as special domains that need not be explicitly declared (Aldrich et al. 2002c; Aldrich and Chambers 2004). These special annotations can be also bound to formal domain parameters. In Section 2.5.1, we discuss how the object graph handles these special annotations.

```
// Implicit OWNER parameter
class Model<V> implements Listener {
   ...
}
class Main {
  domain DOCUMENT, VIEW;
  // Model::OWNER is bound to Main::DOCUMENT
  DOCUMENT Model<VIEW> model;
  ...
}
// vs. explicit OWNER parameter
class Model<OWNER, V> implements Listener {
  ...
}
class Main {
  domain DOCUMENT, VIEW;
  // Model::OWNER is bound to Main::DOCUMENT
  Model<DOCUMENT, VIEW> model;
  ...
}
```

Figure 2.10: Listeners: using the `OWNER` keyword.

2.3.3.1 `OWNER`

Each class has an implicit domain parameter that need not be declared and is named `OWNER`. The `OWNER` implicit parameter always occurs as the first element in the list of domain parameters of a class. Fig. 2.10 shows equivalent annotations that make the implicit `OWNER` parameter explicit.

2.3.3.2 `shared`

Objects can be marked with the `shared` annotation to indicate that they may be aliased globally. But `shared` references may not alias non-`shared` references. Typically, `shared` references are needed for static fields, all of which may refer to aliases that are not related to any object instance. In most cases, the use of static fields is discouraged. In general, the use of `shared` is under the control of the developer, and she could avoid using `shared` altogether, since `shared` is mainly designed to inter-operate with legacy code or third-party libraries. We often use the `shared` annotation for immutable objects like `String` objects.

Nevertheless, `shared` introduces a gap in reasoning about communication integrity. It is not the only one, however. For instance, calls to native methods are another. As a result, external coding guidelines may be needed to discourage the liberal use of the `shared` annotation.

2.3.3.3 `unique`

The annotation `unique` indicates an object to which there is only one reference, such as a newly created object. An object marked `unique` can be passed linearly from one domain to another.

2.3.3.4 `lent`

One ownership domain can temporarily lend an object to another domain and ensure that the second domain does not create a persistent reference to the object, e.g., by storing it in a field. Such an object has the annotation `lent`.

2.4 Static Analysis

A static analysis extracts from an annotated program a *global* object graph that uses object hierarchy to convey architectural abstraction. I explain the static analysis by discussing the following representations of an object-oriented program:

- The Type Graph or TGraph (Section 2.4.1) represents the type structure, and is similar to a class diagram, enhanced with information about the ownership domain annotations;
- The Object Graph or OGraph (Section 2.4.2) represents the object structure and is similar to an object diagram;
- The Display Graph or DGraph (Section 2.4.3) is the object graph with which the developer interacts, to control the abstraction by ownership hierarchy and by types, as well as the level of visual detail.

2.4.1 Type Graph

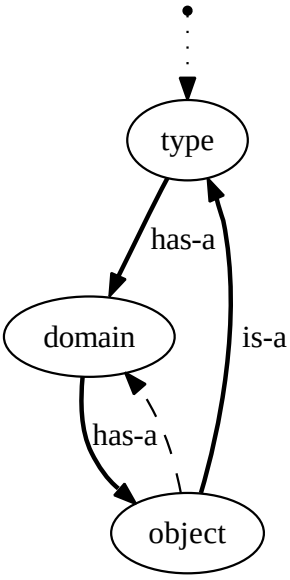
The Type Graph or TGraph represents the type structure of the objects that the code manipulates. A type graph can be considered a kind of UML class diagram that also shows ownership domain annotations, including formal domain parameters. One can build a Type Graph using an implementation of the Visitor design pattern (Gamma et al. 1994, p. 331), to traverse the Abstract Syntax Tree (AST) of an annotated program (Fig. 2.11(a)).

In the type graph, a type declared in the program has domains declared in it. Each local or formal domain declaration has field declarations. In turn, a field declaration has a declared type. But because these types are shared, the type graph is non-hierarchical.

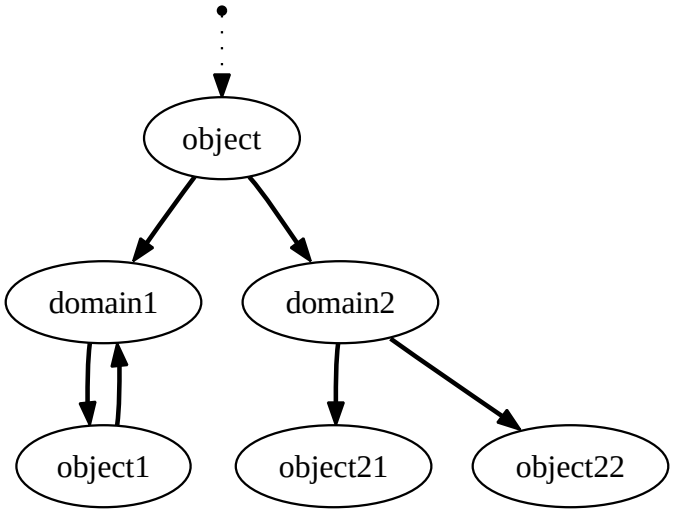
Fig. 2.12 shows the type graph for the Listeners system. A white-filled solid-border box represents a type. A white-filled dotted-border box represents a formal domain parameter, e.g., `M`, declared inside a type. A white-filled dashed-border box represents an actual domain, e.g., `DOCUMENT`. A grey-filled box represents a field declaration inside a domain. A thick dotted edge represents a type relationship. A solid edge represents a field reference.

A type graph is inadequate as a runtime architecture for the following reasons.

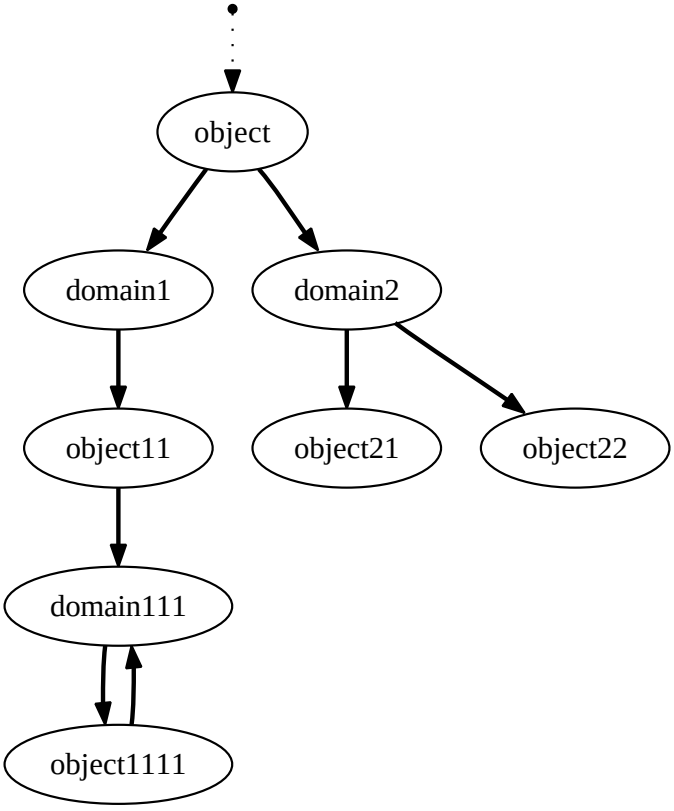
A type graph does not show a hierarchy of objects and domains. In a type graph, a field declaration does not have children objects. Rather, a field declaration has a type, a type has domains, and a domain has other field declarations. For example, the field declaration `barChart` has type `BarChart`, and the type `BarChart` has the formal domain parameter `M` and the actual domain `OWNED`. In turn, the domain declaration `OWNED` contains the field declaration `listeners`. Thus, in a type graph, one cannot view the children of an object without going through its declared type.



(a) Type Graph (TGraph).



(b) Object Graph (OGraph).



(c) Display Graph (DGraph).

Figure 2.11: Relation between Type Graph, Object Graph and Display Graph.

A type graph does not reflect possible aliasing. The ownership domain type system guarantees that two objects in different domains can never alias. But two objects of compatible types, in the same domain, may alias. E.g., if `DOCUMENT` has a field declaration `lstnr` of type `Listener`, it may refer to the same object as the field declaration `model` of type `Model`, because `Model` is a subtype of `Listener`.

If two objects may alias, an object graph conservatively shows them as one. In general, merging objects based on only the aliasing precision provided by the ownership domain type system could yield imprecise results. For example, one could use an intra-domain alias analysis to better approximate the set of objects that may alias at runtime. But experience in applying the analysis on real object-oriented code confirms that the annotations give more than enough precision about aliasing, as long as most object references are declared—or instantiated—with precise types, instead of `java.lang.Object` (Section 2.4.3.2 (Page 59) discusses the difference between using declarations and object allocations). In fact, in most object graphs, one may need to further abstract objects in a domain, based on their declared types (Section 2.4.3.2, Page 59).

In practice, to avoid merging all objects in a domain that have a raw type such as `List`[5], we suggest but do not require refactoring the code to use a generic type, say `List<String>`.

In a type graph, a domain declaration does not directly show all the objects that are in a given domain. The type graph contains field declarations only for the locally declared fields. For instance, the type `List<Listener>` declares its `obj:Listener` field in the `ELTS` domain parameter on `List`. Such fields do not appear where the actual domain is declared. Hence, in the type graph, the formal domain parameter `M` inside `BarChart` is empty, even though it is bound to the `ELTS` on `List` (Fig. 2.12).

A type graph shows formal domain parameters, which do not exist at runtime. Parametric library code often creates interesting architectural relationships in application objects, when these parameters are bound to the specific domains on specific objects created by the application at runtime. So, a static analysis must resolve these parameters to ensure that the relevant object relations appear at the level of the global application object structures.

2.4.2 Object Graph

The analysis computes an Object Graph or OGraph, which soundly approximates any true Runtime Object Graph (ROG) (Fig. 2.11(b)). An OGraph is a graph with two types of nodes, OObjects and ODomains. Edges between OObjects correspond to field reference points-to relations. The root of the graph is a top-level ODomain. For now, assume that the nodes form a hierarchy[6], where each OObject node has a unique parent ODomain, and each ODomain node a unique parent OObject (Fig. 2.13). We will refine later the ODomain and OObject data types.

[5]Generic types were introduced to Java as of version 1.5. Raw types are still part of Java, mostly for backwards compatibility with earlier code bases. We believe that most older Java code is being migrated to use generic types. Indeed, refactoring to generics has mature tool support in Eclipse (Fuhrer et al. 2005). So the overall trend is for more precise declared types in Java code.

[6]In fact, a graph of ODomains and OObjects can have cycles, as we discuss in Section 2.4.2.3.

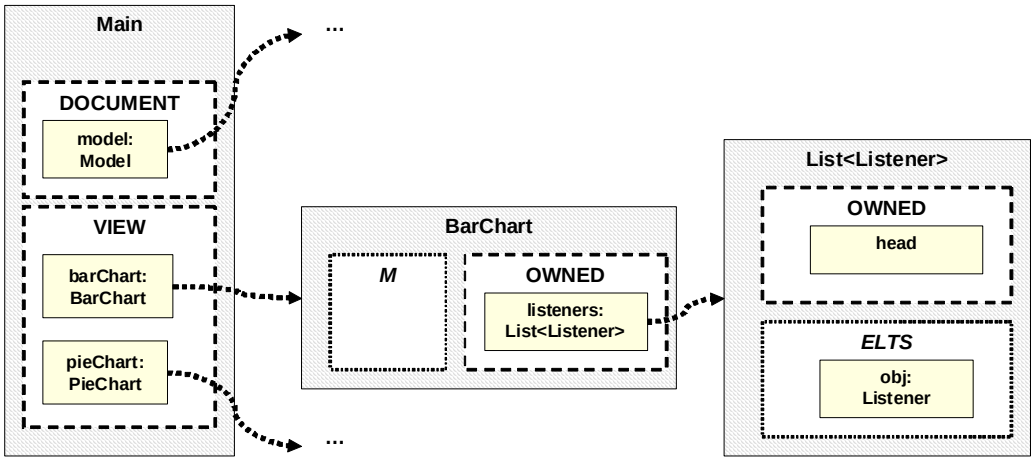


Figure 2.12: Listeners: *type graph*.

$$
\begin{aligned}
G \in \mathsf{OGraph} \quad & ::= (\textbf{Objects} = \{O \ldots\},\ \textbf{Domains} = \{D \ldots\},\ \textbf{Edges} = \{E \ldots\}\ ) \\
& ::= (PtO, PtD, PtE) \\
D \in \mathsf{ODomain} \quad & ::= (\ \textbf{Id} = D_{id}, \textbf{Parent} = O_{id}, \textbf{Domain} = d\ ) \\
& ::= (\ D_{id}, O_{id}, d\ ) \\
O \in \mathsf{OObject} \quad & ::= (\ \textbf{Id} = O_{id},\ \textbf{Parent} = D_{id}, \textbf{Type} = C\ ) \\
& ::= (\ O_{id},\ D_{id}, C\ ) \\
E \in \mathsf{OEdge} \quad & ::= (\ \textbf{From} = O_{src}, \textbf{Field} = f,\ \textbf{To} = O_{dst}\ ) \\
& ::= (\ O_{src}, f,\ O_{dst}\ )
\end{aligned}
$$

Figure 2.13: Initial data type declarations for the OGraph. The formal to actual bindings are not shown.

2.4.2.1 Overview

At a high level, the analysis distinguishes between objects in different domains, and abstracts objects to pairs of domains and types. The analysis adopts the following approach to possible aliasing: in a given domain, two field declarations with compatible types are merged. The analysis also substitutes actual domains to formal domain parameters. To do so, the analysis maintains a set of formal to actual bindings (not shown in Fig. 2.13). Finally, the analysis adds edges between objects.

Object merging. Different executions may generate a different number of objects at runtime, for instance of `BarChart` objects. But the static object graph must represent all possible executions. To address this, the object graph abstracts multiple runtime objects with a canonical object. Further, exactly one canonical object in the object graph represents each object in a ROG.

```
1  class Main {
2    domain DOCUMENT, VIEW;
3
4    DOCUMENT Model<VIEW> model1 = new Model<VIEW>();
5    DOCUMENT Model<VIEW> model2 = new Model<VIEW>();
6
7    VIEW Model<DOCUMENT> model3 = new Model<DOCUMENT>();
8    model3 = model1; // Illegal assignment
9
10   DOCUMENT Model<DOCUMENT> model4 = new Model<DOCUMENT>();
11   model4 = model1; // Illegal assignment
12 }
```

Figure 2.14: Listeners: possible aliasing.

Object aliasing. The object graph maintains an aliasing invariant, i.e., no one runtime object appears as two different canonical objects in the graph. To enforce this invariant, the analysis relies on the ownership domain annotations that give some precision about aliasing, without requiring an alias analysis. The type system guarantees that two objects in different domains cannot alias. But two objects in the same domain may alias. So, the analysis merges two field declarations *in the same domain*, if their types are related by inheritance.

For example, consider the following variation on the Listeners example (Fig. 2.14). The OGraph represents the two object allocations `model1` and `model2` in the same domain `DOCUMENT` into one OObject. On the other hand, the analysis creates a separate OObject for `model3` since it is in the different domain `VIEW`.

Although `model4` is also of type `Model` and is in the `DOCUMENT` domain, it takes different domain parameters than `model1` or `model2`. Indeed, the type system prevents the assignment of `model4` to `model1`, and vice versa, i.e., these two may not alias. So, the analysis creates a separate OObject for `model4` and does not reuse the one for `model1` or `model2`.

Domain parameters. Formal domain parameters do not exist at runtime. As a result, the OGraph does not have formal domain parameters. Instead, the OGraph shows an OObject that the program declares in a formal domain in the corresponding actual ODomain that the formal domain parameter is bound to, starting from the root object. This is important for soundness, because each runtime object that is actually in a domain at runtime must appear in that domain in the object graph. It is as if the analysis *pulls* objects declared inside a formal domain parameter into each actual domain that is bound to the formal domain parameter[7].

2.4.2.2 Abstract interpretation

The static analysis abstractly interprets the program to produce the OObjects, ODomains, and OEdges in the OGraph (Fig. 2.13). The analysis distinguishes between different instances of the same class that are in different domains. In addition, the analysis maintains a mapping from formal domain parameters to the representatives in the OGraph. For reasons we discuss later, the

[7]Previous formalizations of the object graph extraction static analysis accounted for formal domain parameters using an explicit pulling (Abi-Antoun and Aldrich 2007b, 2009a).

analysis generates an OObject in the OGraph when it encounters an object allocation expression, i.e., `new` expression, rather than a variable or field declaration.

Notation. In the following discussion, we use the following notation, to fully qualify objects and domains:

- $obj.DOM$ refers to either a public or a private domain DOM inside object obj, e.g., `main.DOCUMENT`. It effectively treats a domain as a field of an object;
- $obj1.DOM.obj2$ refers to the object $obj2$ inside the domain DOM, e.g., `main.DOCUMENT.model`;
- $fobj \ldots DOM$ refers to a public domain. The ownership domain type system allows path-dependent annotations that are of the form `obj1.obj2...DOM`, where `obj1`, `obj2`, `...`, are chains of final fields or variables, and `DOM` is a public domain declared on the type of the last object in the path;
- $C::d$ refers to a domain d qualified by the class C that declares it.

Example. On the Listeners example, the analysis works as follows (Fig. 2.15). First, the user selects a root type, in this case, the class `Main`. The analysis creates an OObject (O0) for the root object allocation. Then, it analyzes the class `Main` in the context of the (OObject) (O0).

In doing so, the analysis creates two ODomains for the two domains `DOCUMENT` and `VIEW` that `Main` declares, D1 and D2, respectively. For the object allocations inside `Main`, the analysis creates two OObjects `barChart` (O1) and `pieChart` (O2) inside VIEW, and an OObject `model` inside DOCUMENT (O3). Because of the field references, the analysis also creates OEdges from the current object `main` to the newly created objects, E1, E2, and E3.

The analysis then interprets the allocation of a `BarChart` object, by binding the formal domain parameter `BarChart::M` to D1.

In Fig. 2.16, the analysis analyzes the class `BarChart` and its superclass `BaseChart` in the context of the OObject `barChart` and the bindings of formal to actual domains, e.g., that the formal domain parameter `M` is bound to the ODomain `main.DOCUMENT`. While analyzing `BaseChart`, the analysis creates an ODomain for `OWNED` (D3), and an OObject for `List<Listener>` (O4).

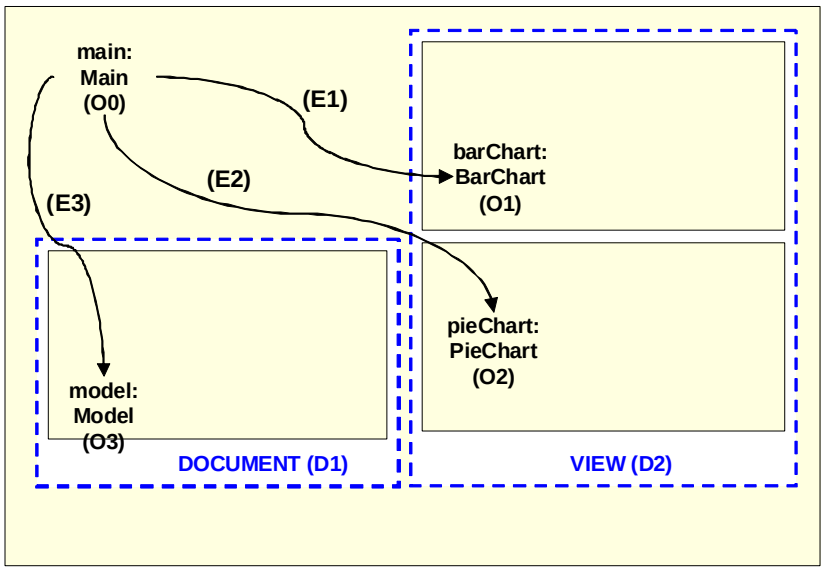
Class `BaseChart` declares a `listeners` field in domain `OWNED`. So the analysis adds an OEdge (E4) from `barChart` to `listeners` inside its `OWNED` domain. Note for example that analysis does not add an edge from `barChart` to `listeners` inside `pieChart`.

Next, the analysis analyzes the class `List<Listener>` in the context of the OObject `listeners` and the bindings in scope. When interpreting the virtual field declaration inside `List`, the analysis looks up all the OObjects in the domain `main.DOCUMENT` the types of which are subtypes of `Listener`. For instance, the analysis finds OObject `model`. So, it creates an OEdge from the OObject corresponding to the current OObject `listeners` to that OObject (E5). Note that the analysis does not add an edge from `barChart`'s `listeners` to `pieChart` in `VIEW`, even though `PieChart` also implements the `Listener` interface. As a result, the edges in an OOG are more precise than super-imposing associations from a class diagram.

The analysis of `PieChart`, its superclass `BaseChart`, and `List` is similar to that of `BarChart` and `BaseChart`, and is not shown.

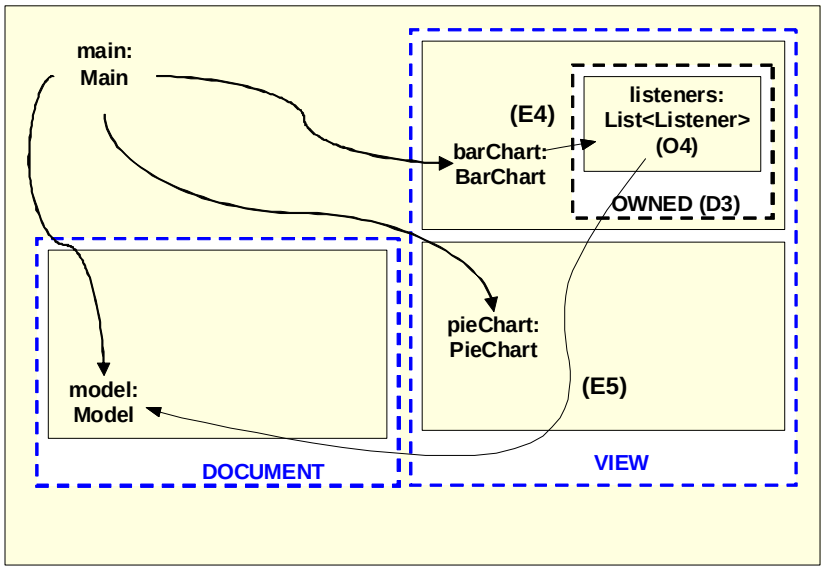
In Fig. 2.17, the analysis processes the class `Model` in the context of the OObject `model`. The analysis creates an ODomain for `OWNED` (D4), an OObject for `List<Listener>` (O5), and an OEdge (E6), then analyzes the object allocation of `listeners`. The analysis then processes the class `List` in the context of the OObject `main.DOCUMENT.model.OWNED.listeners`. The analysis looks up any OObject of type `Listener` in the domain `main.VIEW`, and finds two such OObjects. So it adds an OEdge from the OObject `listeners` to `barChart` (E7), and another from `listeners` to `pieChart` (E8).

The final object graph for listeners is in Fig. 2.18. The root object of an OOG is often an instance of a class that declares the top-level domains and the objects inside them. For readability, we sometimes elide the root domain and the root object from an OOG and consider the domains inside the root type as the top-level domains (Fig.2.19).



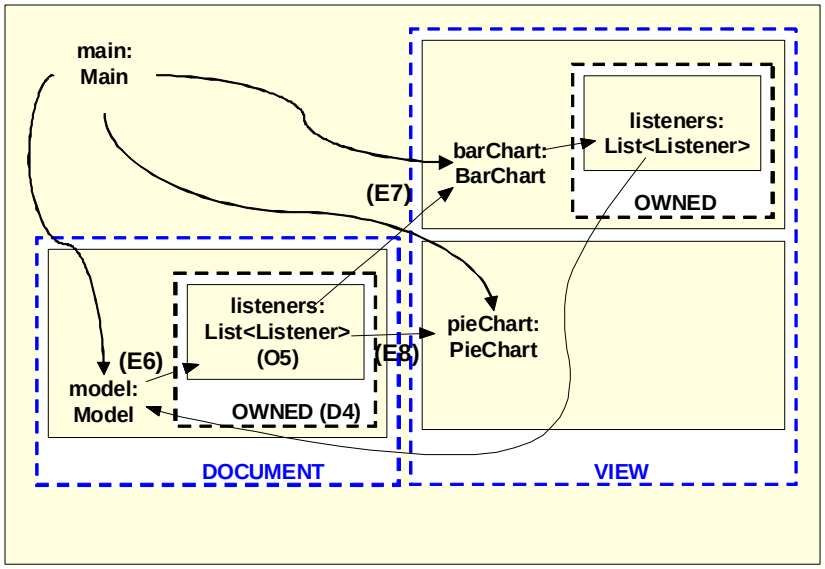

```
1  OObject(main, NULL, Main)
2  Main main = new Main();
3  analyze(main, [])
4  class Main {
5    domain DOCUMENT, VIEW;
6    ODomain(DOCUMENT, main) (D1)
7    ODomain(VIEW, main) (D2)
8    OObject(main.VIEW.barChart, main.VIEW, BarChart) (O1)
9    OEdge(main, main.VIEW.barChart) (E1)
10   VIEW BarChart<DOCUMENT> barChart = new BarChart<DOCUMENT>();
11   analyze(barChart, [BarChart::M ↦ main.DOCUMENT, BarChart::OWNER ↦ main.VIEW])
12   OObject(main.VIEW.pieChart, main.VIEW, PieChart) (O2)
13   OEdge(main, main.VIEW.pieChart) (E2)
14   VIEW PieChart<DOCUMENT> pieChart = new PieChart<DOCUMENT>();
15   analyze(pieChart, [PieChart::M ↦ main.DOCUMENT, PieChart::OWNER ↦ main.VIEW])
16   OObject(main.DOCUMENT.model, main.DOCUMENT, Model) (O3)
17   OEdge(main, main.DOCUMENT.model) (E3)
18   DOCUMENT Model<VIEW> model = new Model<VIEW>();
19   analyze(model, [Model::V ↦ main.VIEW, Model::OWNER ↦ main.DOCUMENT])
20   ...
21 }
```

Figure 2.15: Abstractly interpreting the program, starting with the root class `Main`.




```
1  this ↦ main.VIEW.barChart
2  [BarChart::M ↦ main.DOCUMENT, BarChart::OWNER ↦ main.VIEW]
3  class BarChart<M> extends BaseChart<M> {
4    analyze(barChart, [BaseChart::M ↦ main.DOCUMENT, BaseChart::OWNER ↦ main.VIEW])
5  }
6  this ↦ main.VIEW.barChart
7  [BaseChart::M ↦ main.DOCUMENT, BaseChart::OWNER ↦ main.VIEW]
8  class BaseChart<M> implements Listener {
9    domain OWNED;
10   ODomain(OWNED, main.VIEW.barChart) (D3)
11   OObject(main.VIEW.barChart.OWNED.listeners, main.VIEW.barChart.OWNED, List<Listener>) (O4)
12   OEdge(main.VIEW.barChart, main.VIEW.barChart.OWNED.listeners) (E4)
13   OWNED List<M Listener> listeners = new List<M Listener>();
14   analyze(main.VIEW.barChart.OWNED.listeners,
15     [List::ELTS ↦ main.DOCUMENT, List::OWNER ↦ main.VIEW.barChart.OWNED])
16 }
17 this ↦ main.VIEW.barChart.OWNED.listeners
18 [List::ELTS ↦ main.DOCUMENT, List::OWNER ↦ main.VIEW.barChart.OWNED]
19 T = Listener
20 class List<ELTS T> {
21   OObject(main.DOCUMENT.model, main.DOCUMENT, Model) ∈ lookup(main.DOCUMENT, Listener)
22   OEdge(main.VIEW.barChart.OWNED.listeners, main.DOCUMENT.model) (E5)
23   ELTS T obj;
24  }
```

Figure 2.16: Abstractly interpreting the program (continued): `BarChart`, `BaseChart` and `List`.




```
1  this ↦ main.DOCUMENT.model
2  [Model::V ↦ main.VIEW, Model::OWNER ↦ main.DOCUMENT]
3  class Model<V> implements Listener {
4    domain OWNED;
5    ODomain(OWNED, main.DOCUMENT.model)  (D4)
6    OObject(main.DOCUMENT.model.OWNED.listeners, main.DOCUMENT.model.OWNED, List<Listener>)  (O5)
7    OEdge(main.DOCUMENT.model, main.DOCUMENT.model.OWNED.listeners)  (E6)
8    OWNED List<V Listener> listeners = new List<V Listener>();
9    analyze(main.DOCUMENT.model.OWNED.listeners,
10     [List::ELTS ↦ main.VIEW, List::OWNER ↦ main.DOCUMENT.model.OWNED])
11 }
12 this ↦ main.DOCUMENT.model.OWNED.listeners
13 [List::ELTS ↦ main.VIEW, List::OWNER ↦ main.DOCUMENT.model.OWNED]
14 T = Listener
15 class List<ELTS T> {
16   OObject(main.VIEW.barChart, main.VIEW, BarChart) ∈ lookup(main.VIEW, Listener)
17   OEdge( main.DOCUMENT.model.OWNED.listeners, main.VIEW.barChart)  (E7)
18
19   OObject(main.VIEW.pieChart, main.VIEW, PieChart) ∈ lookup(main.VIEW, Listener)
20   OEdge( main.DOCUMENT.model.OWNED.listeners, main.VIEW.pieChart)  (E8)
21   ELTS T obj;
22 }
```

Figure 2.17: Abstractly interpreting the program (continued): `Model` and `List`.

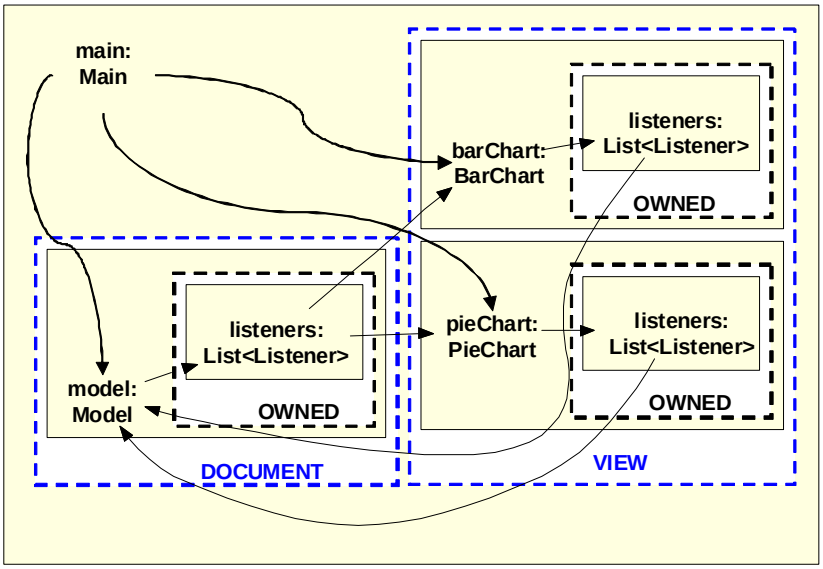


Figure 2.18: Listeners: full object graph, including the root object.

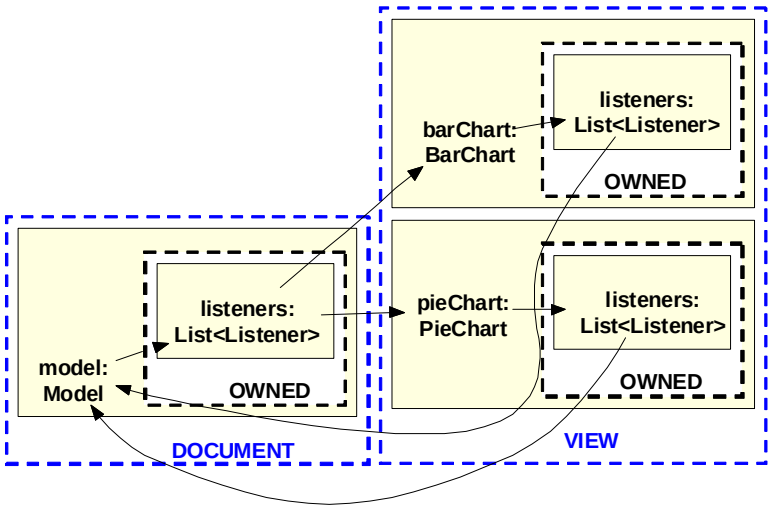


Figure 2.19: Listeners: object graph without the root object and edges from the root.

```
class Main {
  domain OWNED;
  QuadTree<OWNED> aQT = new QuadTree<OWNED>();
}
class QuadTree<M> {
   domain OWNED;
   QuadTree<M> nwQT = new QuadTree<M>();
}
```

Figure 2.20: `QuadTree` with annotations.

2.4.2.3 Recursion

The analysis must handle recursive types which can lead an OGraph to grow arbitrarily deep. For example, consider a class `QuadTree`, which declares fields of type `QuadTree` in its `OWNED` domain (Fig. 2.20). On the `QuadTree` example, the abstract interpretation discussed above would not terminate (Fig. 2.21), as it would keep generating new OObjects and ODomains.

Recursive types. To get a finite OGraph and ensure the analysis terminates, the analysis could stop expanding an OGraph after a certain depth. However, merely truncating the recursion may fail to reveal relations when child objects point to external objects, and the child objects are beyond the visible depth. Instead, the analysis creates a cycle in an OGraph when it reaches a similar context. There are two possible choices (Fig. 2.22).

The first choice is to *unify objects*. For instance, Fig. 2.22(b) shows the resulting OGraph for the `QuadTree` example. Inside `nwQT`, the `OWNED` domain refers back to the same `nwQT` OObject.

The second choice is to *unify domains*. For instance, Fig. 2.22(c) shows the resulting OGraph for the same example. The `OWNED` domain inside `nwQT` is the same as the one inside `aQT`.

We discuss each choice in turn, and why we chose to unify domains in the end.

Unifying objects. Any sound solution to the problem must attempt to always create objects until it detects that it is creating a similar object to one it created before. In that case, the analysis just uses the existing similar object. One can imagine multiple notions of similarity; it can be any equivalence relation, as long as the number of dissimilar objects is finite. For example, one could adopt the following similarity relation between two objects A and B if:

1. A and B are of the same type, including actual domain parameters;
2. A and B came from the same source domain d (not ODomain D – two objects in different d's may end up, after formal to actual substitution, in the same D);
3. A and B are below a depth threshold h; and
4. A and B are transitively inside the same object that is at depth threshold h.

The third condition ensures that the analysis does not unify two objects if one of them is above the threshold, and the fourth condition ensures that the analysis does not add accidental lifted edges by crossing graph boundaries.

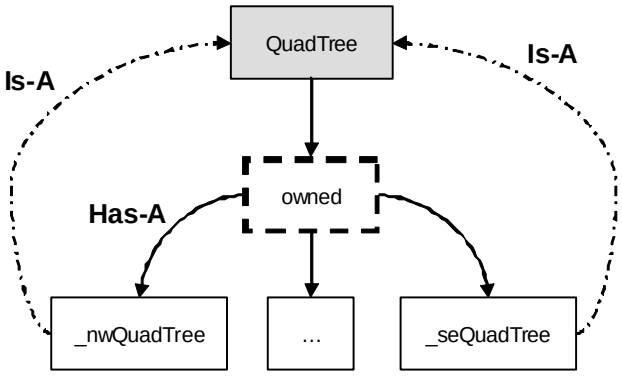
When the analysis does not create an object because it is similar, it still recursively calls the analysis function ($analyze$) on the existing object, because the newly created object could have

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

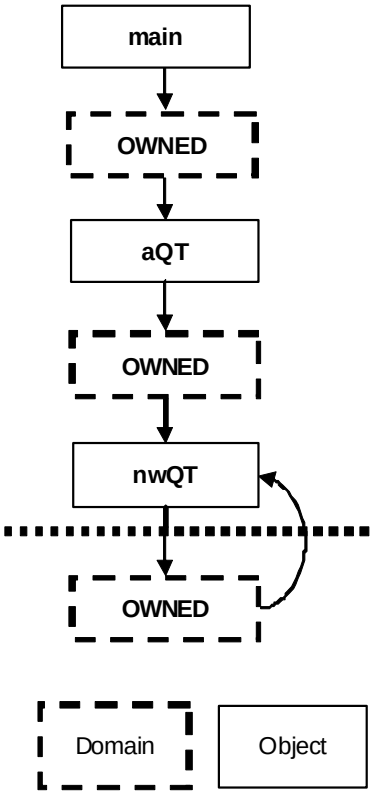
Figure 2.21: `QuadTree` abstract interpretation without cycle detection.

different domain parameters compared to the previous ones, so the recursive call could produce new edges, even ones that show up above the threshold.

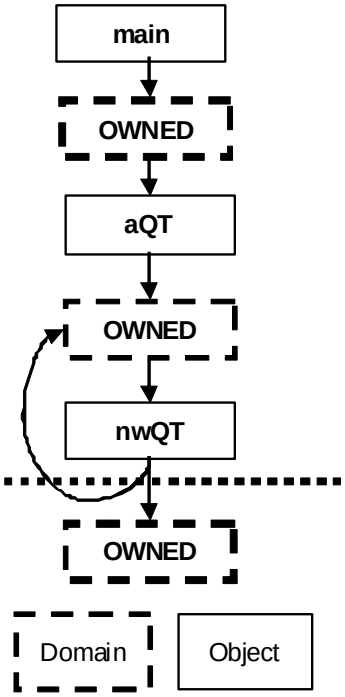
However, unifying objects is problematic. To identify similar objects, it is necessary to detect they have the same owning ODomain. If an ODomain has a unique owning OObject, this becomes circular. Moreover, in order to add edges, we lookup objects in a given domain by their



(a) `QuadTree` *type graph.*



(b) Unifying objects.



(c) Unifying domains.

Figure 2.22: Handling the recursion in `QuadTree`.

type. Since recognizing domains is important, we adopted the solution of unifying domains.

Unifying domains. Instead, unifying domains is less problematic, because it is simpler to recognize when two ODomains have the same underlying domain declaration d.

The analysis creates a cycle in the OGraph when the same ODomain appears as the child of

$$
\begin{aligned}
D \in \mathsf{ODomain} \quad & ::= (\ \textbf{Id} = D_{id},\ \textbf{Domain} = \ C{::}d\ ) \\
& ::= (\ D_{id},\ C{::}d\ )
\end{aligned}
$$

Figure 2.23: Revised data type declaration for ODomain. OObject and OEdge are unchanged.

two OObjects. This justifies an ODomain not having a unique owning OObject, and revising accordingly the data type declaration for an ODomain (Fig. 2.23). We now qualify a domain d by the class C that declares it, for example, `Main::DOCUMENT`. With the revised data structures, the abstract interpretation of `QuadTree` example now terminates (Fig. 2.24).

2.4.2.4 Domain parameters

Recall, the analysis distinguishes between different instances of the same class that are in different domains. We now increase the precision of the analysis and distinguish between instances of the same class in the same domain, that have different actual domain parameters.

Consider a variation on the Listeners example (Fig. 2.25). If we consider that the `List` class takes a domain parameter for its owning domain, `OWNER`, and another for the list elements, `ELTS`, then `List` has type `List<OWNER, ELTS T>`. We want the analysis to distinguish between two `List` object allocations with different actual domains passed in for `OWNER` or `ELTS` (Fig. 2.26). In Chapter 3, we extend the current data type declarations (Fig. 2.23), and additionally include, in an OObject, the actual domain parameters $\overline{D}$, rather than just the owning domain D.

2.4.3 Display Graph

We often do not display an OGraph directly but instead unfold it as a Display Graph or DGraph (Fig. 2.11(c)). The DGraph is the object graph that the tool displays to a developer, and with which the developer interacts.

2.4.3.1 Depth limiting

An OGraph can have cycles. So a DGraph displays an OGraph by unfolding it to a user-specified depth (Fig. 2.27). Increasing the unfolding depth displays more objects. Decreasing the depth collapses the substructure of objects that are already displayed.

In addition, a DGraph adds *lifted edges*[8]to account for any edges in the OGraph below the unfolding depth, using their nearest visible ancestor objects above the unfolding depth. Lifting edges is a well-known technique when visualizing hierarchical representations (Fahmy and Holt 2000).

For instance, for the `QuadTree` example, our visualization shows one `QuadTree` object within another, down to a finite depth (See Fig. 2.28).

[8]**Definition of edge lifting:** If node x has an edge to node y, and x is a descendant of PX and y is a descendant of PY, then we lift the edge (x, y) to (PX, PY) only if PX and PY are distinct nodes and PX is not a descendant or ancestor of PY.

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

Figure 2.24: Revised example with recursive types.

In a DGraph that visualizes an OGraph, there are two ways to reduce the level of detail. One is to restrict the unfolding depth, and another is to expand or collapse the substructures of selected elements.

Edge lifting due to limited unfolding depth. The limited unfolding depth results in the creation of lifted edges. In our implementation, the user interactively controls the unfolding depth.

```
1 class Main {
2   domain OWNED, DOCUMENT, VIEW;
3   ...
4   listViews = new List<Main::OWNED, Main::VIEW Listener>();
5   ...
6   listModels = new List<Main::OWNED, Main::DOCUMENT Listener>();
7 }
```

Figure 2.25: Listeners: distinguishing objects based on domain parameters.

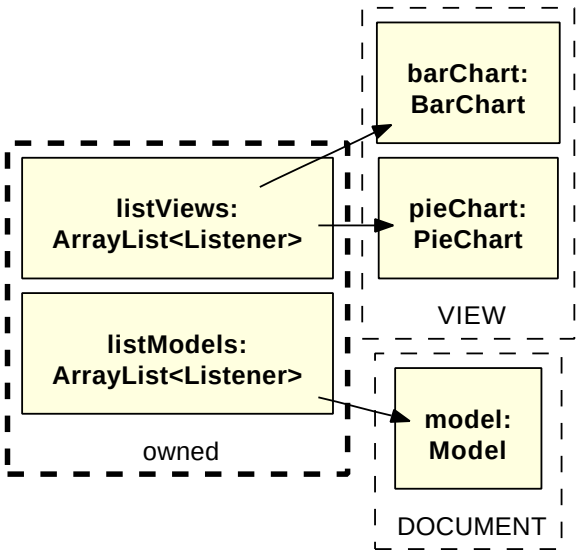


Figure 2.26: Listeners: object graph distinguishing objects based on domain parameters.

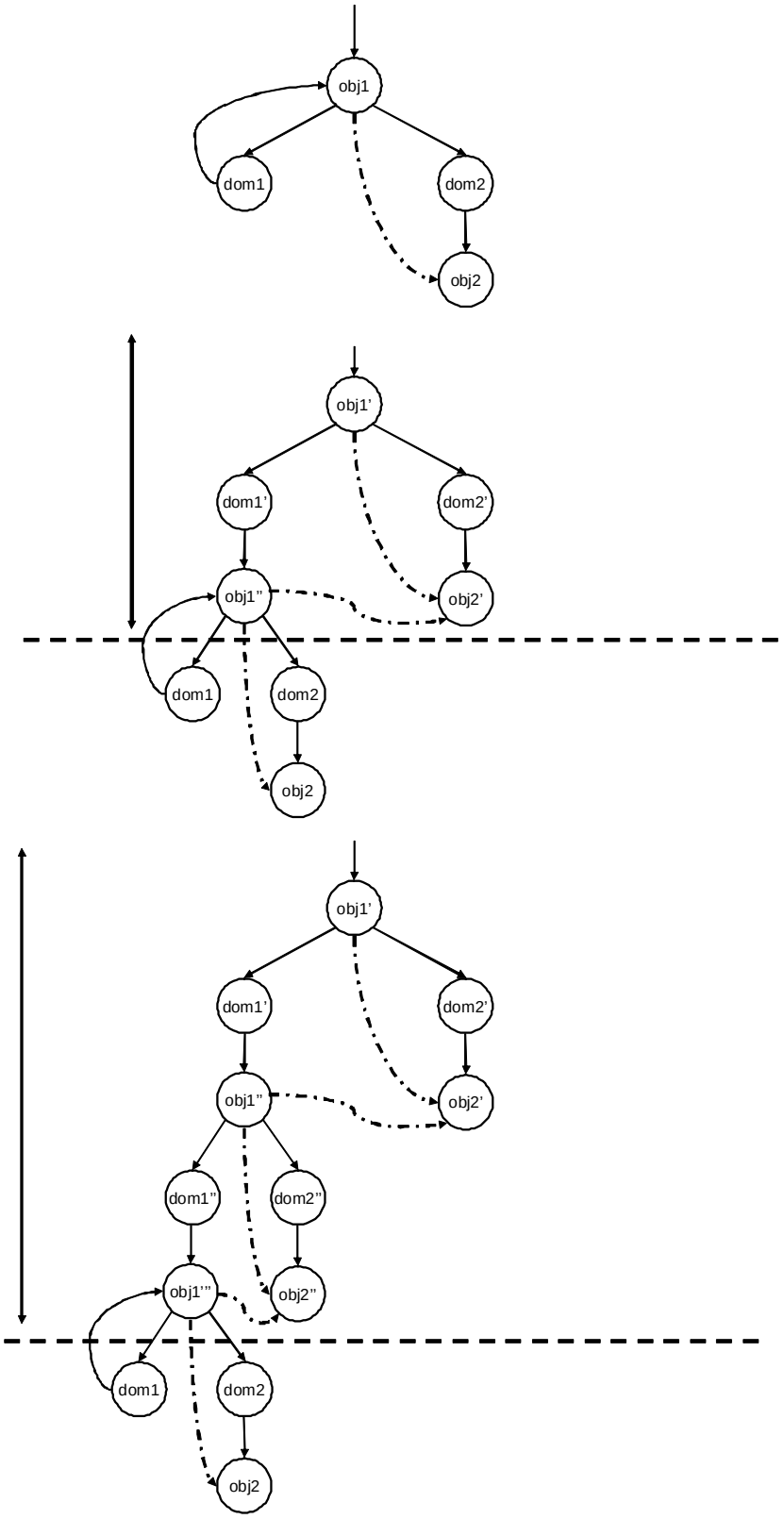
Edge lifting due to collapsing substructures. Edge lifting can also occur when the user expands or collapses individual elements. For example, in `barChart`'s domain `OWNED`, a `listeners` object refers to a `model` object in domain `DOCUMENT` (Fig. 2.29(a)). If the user reduces the unfolding depth, or if she collapses `barChart`'s substructure, the analysis adds a lifted edge from `barChart` to `model` (Fig. 2.29(b)).

2.4.3.2 Abstraction by types

An OOG provides architectural abstraction primarily by ownership hierarchy. In addition, an OOG can abstract objects within each domain by their declared types.

In many object-oriented systems, many types extend from common base classes or implement common interfaces. For instance, both `BarChart` and `PieChart` classes implement the `Listener` interface to realize the Observer design pattern.

Declaration-based view. In keeping with the good practice of programming to an interface instead of an implementation, many field declarations could have interface types. Consider the following variation on the Listeners system, which also declares a field `lstnr` of type `Listener` (Fig. 2.30).



(c)

Figure 2.27: Unfolding an OGraph.

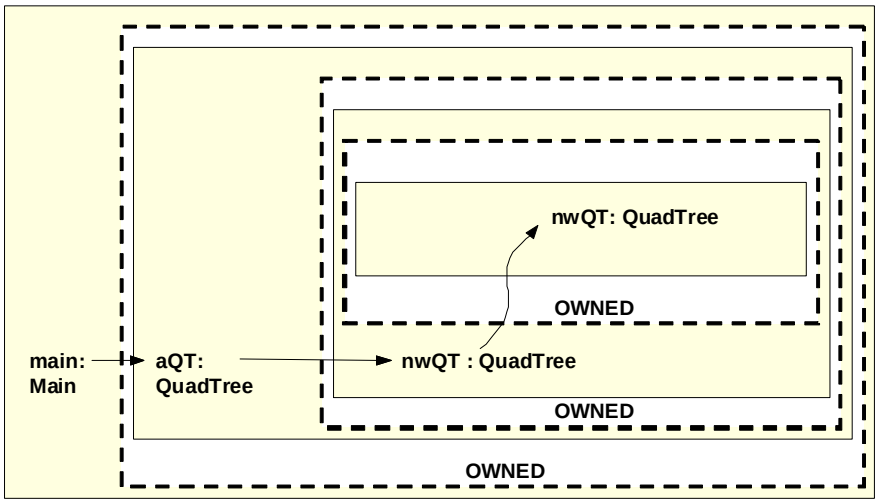
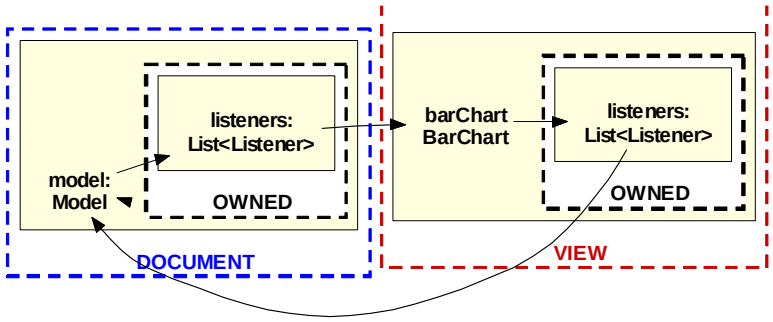
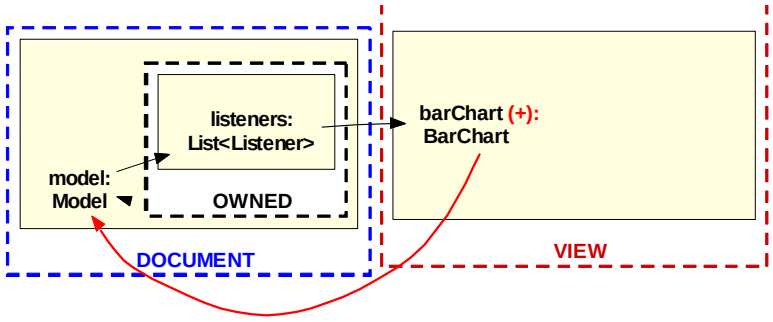


Figure 2.28: `QuadTree` OOG.



(a) Showing `barChart`'s substructure.



(b) Lifted edge between objects `barChart` and `model`.

Figure 2.29: Listeners: limiting the unfolding depth or hiding `barChart`'s substructure adds lifted edges.

The references `barChart`, `pieChart` and `lsntr` are in the same `VIEW` domain. Recall that both `BarChart` and `PieChart` extend `BaseChart`, and `BaseChart` implements the `Listener` interface. As a result, the analysis merges them into the same object in the object graph.

Instantiation-based view. A key insight, however, is that there are no object creations of interface types. So the analysis considers only object creation expressions and generates an

```
1  class Main {
2    domain DOCUMENT, VIEW;
3
4    DOCUMENT Model<VIEW> model = new Model<VIEW>();
5    VIEW BarChart<DOCUMENT> barChart = new BarChart<DOCUMENT>();
6    VIEW PieChart<DOCUMENT> pieChart = new PieChart<DOCUMENT>();
7
8    VIEW Listener lstnr = null;
9    ...
10 }
```

Figure 2.30: Listeners: illustration of interfaces causing merging.

Instantiation-Based View (IBV)[9]. One limitation, however, is that an IBV can be problematic when some code is not available, because it requires knowledge of all the allocation points of objects in the program.

Using an IBV, the type graph would not contain a field declaration `lstnr` of type `Listener`. Rather, it would have field declarations for `barChart` and `pieChart` of type `BarChart` and `PieChart`, respectively. Then, the object graph would show distinct `barChart` and `pieChart` objects, since there is no subtyping relation between their types, `BarChart` and `PieChart`, respectively (Fig. 2.31). In most cases, unless otherwise specified, the analysis will use an Instantiation-Based View[10].

Abstraction. An Instantiation-Based View (IBV) prevents the excessive merging of objects in domains, but may reduce the abstraction and lead to clutter in the object graph. Consider for example the situation where there are many other subclasses of `AbstractChart`, such as `LineChart`, `ColumnChart`, `ScatterChart` and `DoughnutChart`.

Trivial types. To improve abstraction and reduce clutter, an OOG can merge two OObjects in a given ODomain whenever they share one or more least upper bound (LUB) types. The resulting object has an intersection type that includes all the least upper bounds.

In Java-like languages, every class inherits from `java.lang.Object`. However, merging all the OObjects in a domain into a single OObject of type `Object` would result in a sound but uninteresting OOG. So the heuristic does not merge OObjects that have types that share only trivial types as supertypes. Trivial types are user-configurable and typically include `java.lang.Object`, `Cloneable` and `Serializable` from the Java Standard Library. Many marker interfaces that do not declare any methods, such as `RandomAccess`, are good candidates to be included in the list.

Applying abstraction by trivial types on the raw Listeners OOG (Fig. 2.31) produces an OOG that is less cluttered (Fig. 2.32). In particular, the OOG now merges all the chart objects in the `VIEW` domain into one object, shown as `listener:Listener`.

[9]This is similar to how (Bacon and Sweeney 1996) use Rapid Type Analysis (RTA) to determine a method call's receiver during call graph construction. However, RTA alone is insufficient. (Rayside et al. 2005) proposed a static object graph analysis based on RTA which produced trivial over-approximations for most programs.

[10]For simplicity, I often omit the explicit object allocations in the sample programs included in this document.

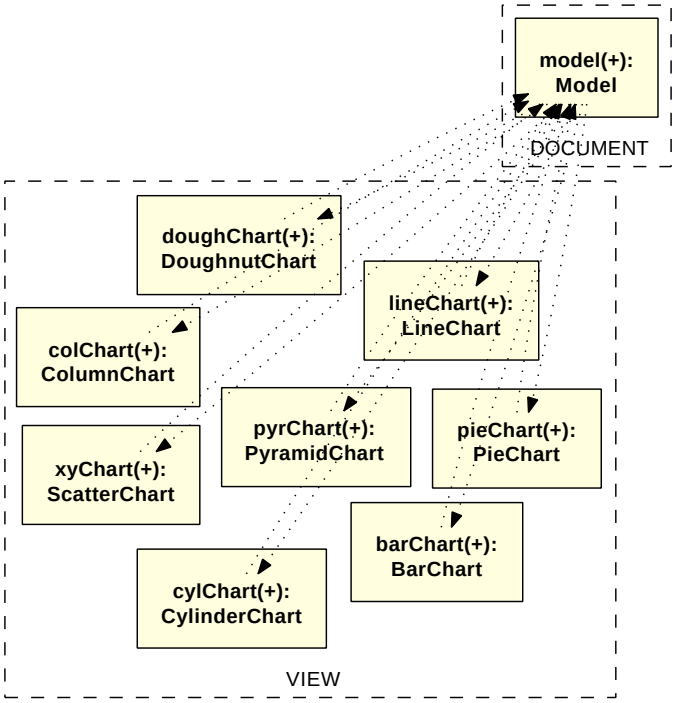


Figure 2.31: Listeners: Instantiation-Based View (IBV).

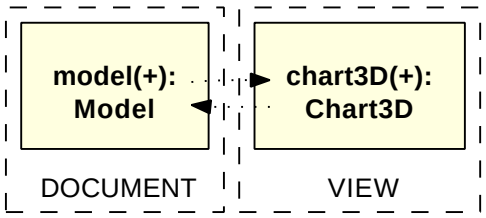


Figure 2.32: Listeners: abstraction by trivial types.

Design intent types. Abstraction by trivial types can quickly unclutter an OOG but is not very precise. Assume that the developers distinguish between two- and three-dimensional charts in the type hierarchy, and define a `Chart2D` and a `Chart3D` interface. Classes such as `LineChart`, `ColumnChart`, `ScatterChart` and `DoughnutChart` implement a `Chart2D` interface. Other classes such as `CylinderChart` and `PyramindChart` implement the `Chart3D` interface. Finally, some classes implement both interfaces (Fig. 2.33).

Similarly, we may want the OOG to distinguish between two- and three-dimensional charts. In particular, we may want to treat 3D charts as more architecturally significant than 2D charts.

For this purpose, the developer defines a list of design intent types, ordered from most to least architecturally relevant. For instance, she adds the interfaces `Chart3D` and `Chart2D` to the list, in that order. When determining the object with which to merge the OObject for `pieChart:PieChart`, the analysis finds the first type in the list of design intent types that is a supertype of `PieChart`. For instance, the analysis picks `Chart3D`. So it collapses several OObjects into a DObject of type `Chart3D`. Then, the analysis finds the first type in the list of design intent types that is a supertype of `PyramidChart`. In this case, the analysis picks `Chart3D` again.

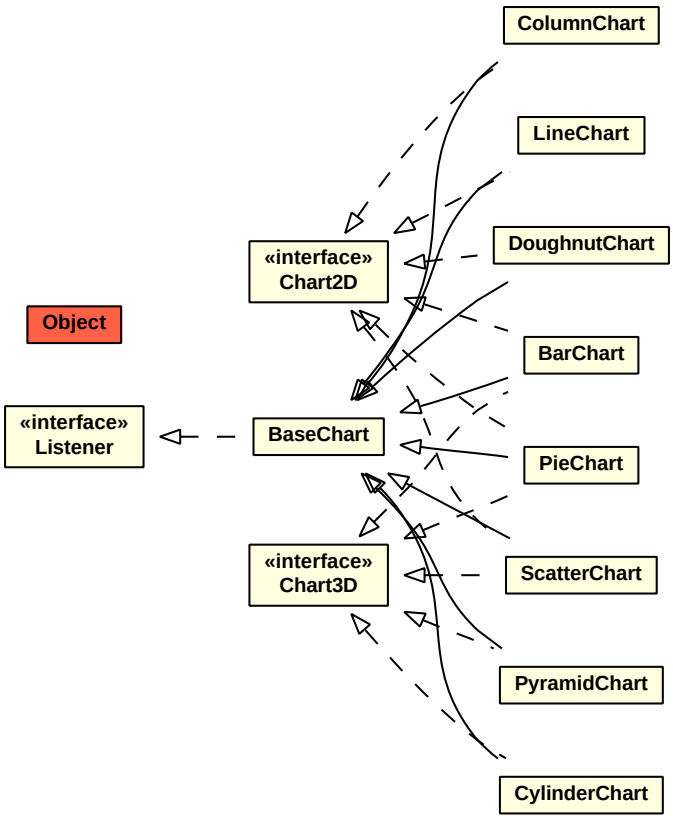


Figure 2.33: Listeners: inheritance hierarchy.

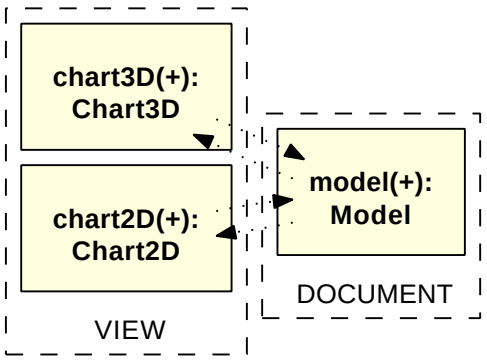


Figure 2.34: Listeners: abstraction by design intent types.

When the analysis processes the field declaration for `doughChart:DoughnutChart`, it picks `Chart2D`. So it creates a DObject of type `Chart2D`. Similarly, it merges `lineChart:LineChart` with `Chart2D`. Applying abstraction by design intent types to the raw Listeners OOG (Fig. 2.31) produces an OOG that conveys the architectural intent of distinguishing between the two kinds of charts (Fig. 2.34).

2.4.4 Summary

An OOG is a graph with two types of nodes, objects and domains. The nodes form a hierarchy where each object node has a unique parent domain and each domain node has a unique parent object. The root of the graph is a top-level global domain. There are two edge types. Edges between objects correspond to field references. Edges between domains correspond to domain links. Compared to previous definitions of object graphs, e.g., (Potter et al. 1998), an OOG explicitly represents clusters of objects using domains and edges between these clusters using domain links. In contrast to other ownership hierarchies (Hill et al. 2002; Potanin et al. 2004), in an OOG, the owner of an object is a domain instead of another object. The ability to define multiple domains per object is useful for modeling multiple architectural tiers in an application. In addition, an OOG supports two forms of hierarchy: strict encapsulation and logical containment. Previous ownership systems which had multiple contexts per object, e.g., (Clarke 2001), support only strict encapsulation, which cannot express many object-oriented design idioms. In contrast, the expressiveness of logical containment makes it easier to both add annotations to existing code as well as control the architectural abstraction in an object graph.

2.5 Advanced Features

We now discuss several additional features.

2.5.1 Displaying objects with special annotations

Objects that have one of the special annotations (`unique`, `lent`, or `shared`) require special handling.

2.5.1.1 `shared` objects

The object graph analysis assumes that all objects marked as `shared` are in one domain. As a result, due to merging objects for soundness, the analysis may excessively merge objects that are in the `shared` domain. Unless the user requests otherwise, we often purposely do not display the `shared` domain in an OOG. Displaying the `shared` domain would be trivial, but would add many uninteresting edges to the OOG. Strictly speaking, excluding the `shared` domain makes the resulting OOG unsound, but we believe it to be an acceptable compromise.

2.5.1.2 `unique` objects

An OOG may not reflect an object marked `unique` until it is assigned to a specific domain. When an object is created, it is `unique`. An inter-procedural flow analysis is needed to track each object from its creation until its assignment to a specific domain. Since the current tool does not implement such a flow analysis, a developer must manually annotate a `unique` object returned from a factory method with the domain in which it should be displayed.

2.5.1.3 `lent` objects

Objects annotated with `lent` are currently missing from the OOG. To display them in the OOG, a flow analysis is needed to determine the domain that a `lent` object is really in. Currently, the workaround is s to manually resolve the `lent` annotation, and to use the more precise annotation.

2.6 Discussion

2.6.1 Assumptions

The SCHOLIA extraction static analysis makes the following assumptions:

- **Sources available:** The program's whole source code and portions of external libraries that are in use have annotations that typecheck[11];
- **Single entry point:** The program operates by creating a main object.
- **Summarized external entities:** Reflection, dynamic code loading or native calls may introduce unknown objects and edges into the system. The annotation system can summarize these external entities using "virtual" or "ghost" (Flanagan et al. 2002) field annotations. The latter are also useful when the sources are unavailable.

2.6.2 Alternate Annotations

There are multiple ways to annotate a program. Fig. 2.35 shows an alternate set of annotations for the Listeners system and the resulting OOG (Fig. 2.36). In these annotations, the `listeners` collection object are no longer in private domains. This allows a client program to modify the `listeners` collection objects directly, which the client could not do if these objects were strictly encapsulated in private domains.

These annotations make the `listeners` list objects appear in the top-level domains and illustrate the potential loss of precision due to merging objects within a domain by their declared types and their domain parameters (in this case, the domain parameters for `BarChart` and `PieChart` are bound to the same domain). For instance, the `listeners` of `pieChart` and those of `barChart` are merged in the `VIEW` domain (Fig. 2.36). However, this object graph is still more precise than a class diagram, which also abstracts objects by type, because *two objects that are in two different domains can never be aliased*. For instance, the analysis can still distinguish between the `listeners` of `model` from those of `pieChart` and `barChart`.

Moreover, a developer can prevent unwanted merging by placing two objects that should never get merged in separate domains. For instance, even if a developer does not wish to use strict encapsulation, she can define public domains, and place the `listeners` objects in public domains (See Fig. 2.37). The resulting object graph is in Fig. 2.38.

The main difference between the OOG in Fig. 2.38 and the one in Fig. 2.3(a) is that, in the former, `LISTENERS` domains appear with a thin dashed border, whereas `OWNED` appears with a

[11] Our static analysis is similar to an Andersen-style points-to static analysis (Andersen 1994). An object-sensitive analysis, e.g., (Milanova et al. 2005), would have this same limitation, because it requires knowledge of all the allocation points of objects in the program.

```
interface Listener {
}
class BaseChart<M> implements Listener {
  OWNER List<M Listener> listeners;
}
class BarChart<M> extends BaseChart<M> {
}
class PieChart<M> extends BaseChart<M> {
}
class Model<V> implements Listener {
   OWNER List<V Listener> listeners;
}
...
class Main {
  domain DOCUMENT, VIEW;
  DOCUMENT Model<VIEW> model;
  VIEW BarChart<DOCUMENT> barChart;
  VIEW PieChart<DOCUMENT> pieChart;
}
```

Figure 2.35: Listeners: alternate annotations.

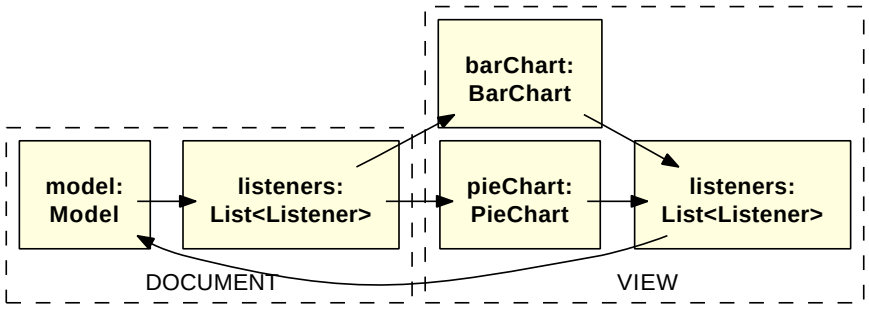


Figure 2.36: Listeners: object graph based on the alternate annotations.

thick dashed border. Recall that a thick dashed border indicate that these instances are owned or strictly encapsulated by their outer objects. And a thin border indicates logical containment. In particular, when using logical containment, a developer can define a public method that returns an alias to a field in a public domain.

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

```
interface Listener {
}
class BaseChart<M> implements Listener {
  public domain LISTENERS; // Public domain
  LISTENERS List<M Listener> listeners;

  // A public method can return a reference to an object in a public domain
  public LISTENERS List<M Listener> getListeners() {
    return listeners;
  }
}
class BarChart<M> extends BaseChart<M> {
}
class PieChart<M> extends BaseChart<M> {
}
class Model<V> implements Listener {
  public domain LISTENERS; // Public domain
  LISTENERS List<V Listener> listeners;
}
...
class Main {
  domain DOCUMENT, VIEW;
  DOCUMENT Model<VIEW> model;
  VIEW BarChart<DOCUMENT> barChart;
  VIEW PieChart<DOCUMENT> pieChart;
}
```

Figure 2.37: Listeners: using public domains.

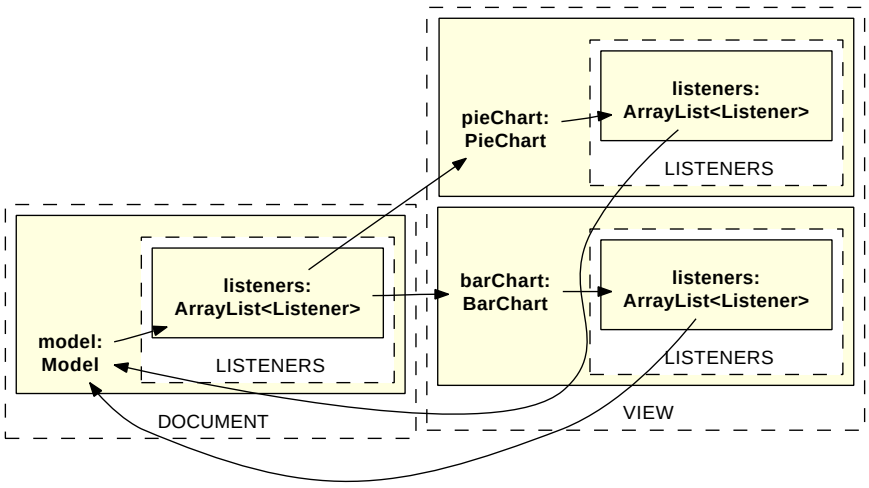


Figure 2.38: Listeners: object graph based on using public domains.

2.6.3 Imprecision

The OOG extraction relies on the type system's guarantee that two objects in different domains cannot be assigned to each other, and thus can never alias. But two objects in the same domain may alias. In the absence of more information about possible aliasing, the analysis can be imprecise in several cases that we discuss next[12].

2.6.3.1 Field assignment in superclass

Consider the code in Fig. 2.39 and the corresponding OOG in Fig. 2.40. The OOG is imprecise because it shows an edge from `c` to `y`, and an edge from `b` to `z`.

In SCHOLIA, a developer can place objects that should not get merged in different domains. Of course, this assumes that the developer is aware of the analysis's bias. For instance, in the above example, the developer adding the annotations can define two domains, `OWNED1` and `OWNED2`, and place `b` and `c` in `OWNED1` and `OWNED2`, respectively (Fig. 2.41). Then, the OOG will not show imprecise edges between `c` and `y`, or `b` and `z` (Fig. 2.42).

[12] Our static analysis is similar to an Andersen-style points-to analysis (Andersen 1994), which has known sources of imprecision, that *object-sensitive* variants address (Milanova et al. 2005). I took these code examples from (Milanova et al. 2005) and annotated them.

```
1  class X {
2    void n() {
3    }
4  }
5  class Y extends X {
6    void n() {
7    }
8  }
9  class Z extends X {
10   void n() {
11   }
12 }
13 class A<P> {
14   P X f;
15   A(P X xa) {
16     this.f = xa;
17   }
18 }
19 class B<P> extends A<P> {
20   B(P X xb) {
21       super(xb);
22   }
23   void m() {
24     lent X xb = this.f;
25     xb.n();
26   }
27 }
```

```
28 class C<P> extends A<P> {
29   C(P X xc) {
30       super(xc);
31   }
32   void m() {
33     lent X xc = this.f;
34     xc.n();
35   }
36 }
37
38 public class Main {
39   domain OWNED;
40   OWNED Y y = new Y();
41   OWNED Z z = new Z();
42   OWNED B<OWNED> b = new B(y);
43   OWNED C<OWNED> c = new C(z);
44
45   public void init() {
46     b.m();
47     c.m();
48   }
49   public static void main(lent String[shared] args) {
50     lent Main system = new Main();
51     system.init();
52   }
53 }
```

Figure 2.39: Field assignment in superclass, adapted from (Milanova et al. 2005).

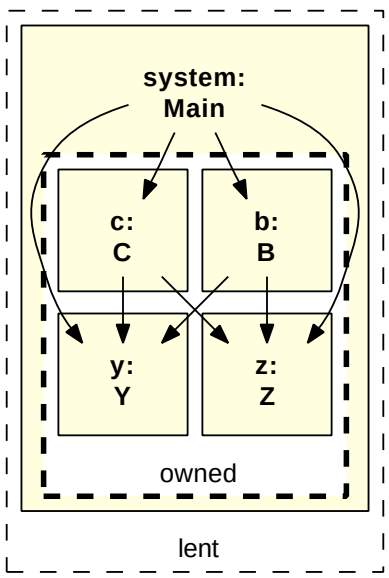


Figure 2.40: Imprecision with field assignment in superclass.

```
class X {
  void n() {
  }
}
class Y extends X {
  void n() {
  }
}
class Z extends X {
  void n() {
  }
}
class A<P> {
  P X f;
  A(P X xa) {
    this.f = xa;
  }
}
class B<P> extends A<P> {
  B(P X xb) {
      super(xb);
  }
  void m() {
    lent X xb = this.f;
    xb.n();
  }
}
class C<P> extends A<P> {
  C(P X xc) {
      super(xc);
  }
  void m() {
    lent X xc = this.f;
    xc.n();
  }
}

public class Main {
  domain OWNED1, OWNED2;
  OWNED1 Y y = new Y();
  OWNED2 Z z = new Z();
  OWNED1 B<OWNED1> b = new B(y);
  OWNED2 C<OWNED2> c = new C(z);

  public void init() {
    b.m();
    c.m();
  }
  public static void main(lent String[shared] args) {
    lent Main system = new Main();
    system.init();
  }
}
```

Figure 2.41: Field assignment in superclass, adapted from (Milanova et al. 2005).

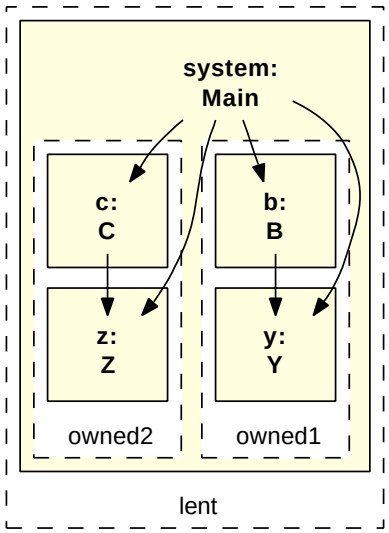


Figure 2.42: Fixing imprecision with field assignment in superclass.

```
1
2  class X {
3  }
4
5
6  class Y {
7  }
8
9
10 class Container<P> {
11   P Object f;
12
13   void put(P Object xa) {
14       this.f = xa;
15   }
16 }
17
```

```
18 public class Main {
19   domain OWNED;
20   OWNED Y y = new Y();
21   OWNED X x = new X();
22   OWNED Container<OWNED> c1 = new Container();
23   OWNED Container<OWNED> c2 = new Container();
24
25   public void init() {
26     c1.put(x);
27     c2.put(y);
28   }
29   public static void main(lent String[shared] args) {
30      lent Main system = new Main();
31      system.init();
32   }
33 }
```

Figure 2.43: Simple code with container, adapted from (Milanova et al. 2005).

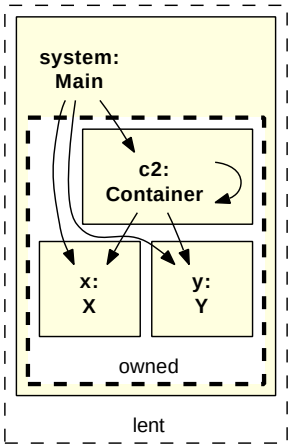


Figure 2.44: Imprecision with container.

2.6.3.2 Imprecision with containers

The use of containers can also cause a precision loss. Consider the code in Fig. 2.43. The corresponding OOG is in Fig. 2.44, and suffers from an imprecision of merging the two `Container` objects. The developer can also prevent this merging by placing `c1` and `c2` in separate domains.

2.7 Summary

To provide architectural abstraction, an object graph must distinguish between objects that are architecturally relevant from those that are not. An OOG provides architectural abstraction primarily by ownership hierarchy, by pushing low-level objects underneath more architectural objects. Thus, only architecturally relevant objects appear in the top-level domains. In turn, each one of those objects has nested domains and objects that represent its substructure, and so on, until low-level, less architecturally relevant objects are reached.

In addition, an OOG can provide abstraction by types, by merging objects in each domain based on their declared types in the program, the notion of subtyping, and optional developer input to specify the architecturally relevant types.

Indeed, collapsing many nodes into one is a classic approach to shrink a graph, and has previously been used in extracting views of the code architecture (Müller and Klashinsky 1988; Kazman and Carrière 1999). However, an OOG is unique in collapsing objects, statically, based on their ownership and type structures, and not according to where they are syntactically declared in the program, some naming convention or a graph clustering algorithm.

Our empirical evaluation in Chapter 4 will confirm that abstraction by ownership hierarchy and by types can reduce the number of objects at the top level by an order of magnitude, compared to a flat object graph. Before we evaluate the analysis in practice on real object-oriented code, we describe it formally and prove key soundness theorems in Chapter 3.

Chapter 3

Formalization of the Object Graph Extraction[1]

In this chapter, I formally describe the static analysis that SCHOLIA uses to extract a hierarchical object graph from a program with ownership domain annotations, and prove key soundness theorems.

The formalization of the static analysis assumes a Java-like program with ownership domain annotations. Section 3.1 reviews the formalization of ownership domains using Featherweight Domain Java (FDJ). Section 3.2 formalizes the Object Graph (OGraph). Section 3.3 discusses soundness. Section 3.4 discusses the Display Graph (DGraph) that a developer sees, including applying the optional abstraction by types. I then discuss a few implementation details in Section 3.5, and conclude with a discussion in Section 3.6.

3.1 Annotations (Featherweight Domain Java)

The SCHOLIA annotations implement the ownership domains type system. For completeness, we reproduce here parts of the Featherweight Domain Java (FDJ) type system (Aldrich and Chambers 2004), with some corrections[2] and additional changes we discuss later.

3.1.1 Syntax

Fig. 3.1 shows the syntax of Featherweight Domain Java (FDJ).

- C ranges over class names;
- T ranges over types;
- f ranges over fields;
- v ranges over values;
- e ranges over expressions;

[1] Portions of this chapter appeared in (Abi-Antoun and Aldrich 2009b).

[2] Errata available at: `http://www.cs.cmu.edu/~aldrich/papers/ownership-domains-errata.html`

- x ranges over variable names;
  d over domain names;
- n ranges over values and variable names;
- S ranges over stores;
- ℓ and v range over locations in the store;
- α, β and γ range over formal domain parameters;
- m ranges over method names. As a shorthand, an overbar is used to represent a sequence.
- A store S maps locations ℓ to their contents: the class of the object, the actual ownership domain parameters, and the values stored in its fields.
- $S[\ell]$ denotes the store entry for ℓ.
- $S[\ell, i]$ to denote the value in the ith field of $S[\ell]$.
- Adding an entry for location ℓ to the store is abbreviated $S[\ell \mapsto C{<}\overline{\ell}{>}(\overline{\ell'})]$.
- $\ell \triangleright e$ represents a method body e executing with a receiver ℓ.
- The result of computation is a location ℓ, which is sometimes referred to as a value v.
- The set of variables includes the distinguished variable `this` used to refer to the receiver of a method.
- The fixed class table CT maps classes to their definitions.
- A program, then, is a tuple (CT, S, e) of a class table, a store, and an expression.

We simplify the formal system slightly by treating the first domain parameter of a class as its owning domain. We use a slightly different syntax in the practical system to emphasize the semantic difference between the owner domain of an object and its domain parameters.

Assumptions. The formal model makes the following simplifying assumptions:

- No `lent` or `unique` annotations: (Aldrich et al. 2002c) showed how to integrate them with an ownership type system. We also discussed how the static analysis might handle these special annotations (Section 2.3.3, Page 41);
- No cast or the resulting `error` expressions to handle failed casts: those are part of FDJ, but are not crucial to this discussion.

Auxiliary judgements. The semantics use many auxiliary judgements (Fig. 3.2, 3.3). These definitions are straightforward and in many cases are derived directly from rules in Featherweight Java. The Aux-Public rule checks whether a domain is public. The next few rules define the $domains$, $links$, $assumptions$, and $fields$ functions by looking up the declarations in the class and adding them to the declarations in superclasses. The $linkdecls$ function just returns the union of the $links$ and $assumptions$ in a class, while the $owner$ function just returns the first domain parameter (which represents the owning domain in the FDJ formal system).

The $mtype$ function looks up the type of a method in the class; if the method is not present, it looks in the superclass instead. The $mbody$ function looks up the body of a method in a similar

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

Figure 3.1: Featherweight Domain Java abstract syntax. Source: (Aldrich and Chambers 2004).

way. Finally, the $override$ function verifies that if a superclass defines method m, it has the same type as the definition of m in a subclass.

In the dynamic semantics (Fig. 3.4), when a method expression reduces to a value, the *R-Context* rule propagates the value outside of its method context and into the surrounding method expression. As this rule shows, expressions of the form $\ell \triangleright e$ do not affect program execution, and are used only for reasoning about invariants that are necessary for link soundness.

Congruence rules allow reduction to proceed within an expression in the order of evaluation defined by Java (Fig. 3.5). For example, the read rule states that an expression $e.f$ reduces to $e'.f$ whenever e reduces to e'.

Finally, we did not include some FDJ rules, e.g., link permission rules, which can be found elsewhere (Aldrich and Chambers 2004).

3.1.2 Typing Rules

The FDJ subtyping rules are in Fig. 3.6. The FDJ static semantics are in Fig. 3.7 and in Fig. 3.8.

$$\frac{(\texttt{public domain}\ d) \in \overline{dom}}{public(d)}\ \textit{Aux-Public}$$

$$\frac{\texttt{class}\ C\texttt{<}\overline{\alpha}\texttt{>}}{params(C) = \overline{\alpha}}\ \textit{Aux-Params}$$

$$\frac{\overline{lnk} = \overline{\texttt{link}\ d_c \rightarrow d'_c} \qquad links(C'\texttt{<}\overline{d}\texttt{>}) = \overline{d_s \rightarrow d'_s}}{links(C\texttt{<}\overline{d}, \overline{d'}\texttt{>}) = ([\overline{d}/\overline{\alpha}, \overline{d'}/\overline{\beta}]\ (\overline{d_c \rightarrow d'_c})),\ \overline{d_s \rightarrow d'_s}}\ \textit{Aux-Links}$$

$$\frac{assumptions(C'\texttt{<}\overline{d}\texttt{>}) = \overline{d_s \rightarrow d'_s}}{assumptions(C\texttt{<}\overline{d}, \overline{d'}\texttt{>}) = ([\overline{d}/\overline{\alpha}, \overline{d'}/\overline{\beta}]\ (\overline{\gamma \rightarrow \delta})), \overline{d_s \rightarrow d'_s}}\ \textit{Aux-Assume}$$

$$\frac{}{linkdecls(C\texttt{<}\overline{p}\texttt{>}) = links(C\texttt{<}\overline{p}\texttt{>}) \cup assumptions(C\texttt{<}\overline{p}\texttt{>})}\ \textit{Aux-LinkDecls}$$

$$\frac{}{owner(C\texttt{<}\overline{p}\texttt{>}) = d_1}\ \textit{Aux-Owner}$$

$$\frac{(\ T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\}\ ) \in \overline{md}}{mtype(m, C\texttt{<}\overline{p}\texttt{>}) = [\overline{d}/\overline{\alpha}]\ \overline{T} \rightarrow T_R}\ \textit{Aux-MType1}$$

$$\frac{m\ is\ not\ defined\ in\ \overline{md}}{mtype(m, C\texttt{<}\overline{d}, \overline{d'}\texttt{>}) = mtype(m, C'\texttt{<}\overline{d}\texttt{>})}\ \textit{Aux-MType2}$$

$$\frac{(\ T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\}\ ) \in \overline{md}}{mbody(m, C\texttt{<}\overline{p}\texttt{>}) = [\overline{d}/\overline{\alpha}]\ (\overline{x} : \overline{T},\ e)}\ \textit{Aux-MBody1}$$

$$\frac{m\ is\ not\ defined\ in\ \overline{md}}{mbody(m, C\texttt{<}\overline{d}, \overline{d'}\texttt{>}) = mbody(m, C'\texttt{<}\overline{d}\texttt{>})}\ \textit{Aux-MBody2}$$

$$\frac{(mtype(m, C\texttt{<}\overline{p}\texttt{>}) = \overline{T'} \rightarrow T') \Longrightarrow (\overline{T} = \overline{T'} \wedge T = T')}{override(m, C\texttt{<}\overline{p}\texttt{>}, \overline{T} \rightarrow T)}\ \textit{Aux-Override}$$

$$CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha}, \overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{assumes}\ \overline{\gamma \rightarrow \delta}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \overline{lnk};\ \overline{md};\ \}$$

Figure 3.2: FDJ auxiliary definitions. Adapted from (Aldrich and Chambers 2004). $mbody$ now exposes the types of a method's parameters.

$$\frac{(\texttt{public}_{opt}\ \texttt{domain}\ d) \in \overline{dom} \qquad domains(C\texttt{<}\overline{d}\texttt{>}) = \overline{d'}}{domains(C\texttt{<}\overline{d}, \overline{d'}\texttt{>}) = \overline{\texttt{this}.d}, \overline{d'}}\ \textit{Aux-Domains}$$

$$\frac{}{domains(\texttt{Object<}\alpha_o\texttt{>}) = \emptyset}\ \textit{Aux-Domains-Obj}$$

$$\frac{fields(C'\texttt{<}\overline{d}\texttt{>}) = \overline{T'}\ \overline{f'}}{fields(C\texttt{<}\overline{d}, \overline{d'}\texttt{>}) = ([\overline{d}/\overline{\alpha}, \overline{d'}/\overline{\beta}]\ \overline{T}\ \overline{f}), \overline{T'}\ \overline{f'}}\ \textit{Aux-Fields}$$

$$\frac{}{fields(\texttt{Object<}\alpha_o\texttt{>}) = \emptyset}\ \textit{Aux-Fields-Obj}$$

$$CT(C) = \texttt{class}\ C\texttt{<}\overline{\alpha}, \overline{\beta}\texttt{>}\ \texttt{extends}\ C'\texttt{<}\overline{\alpha}\texttt{>}\ \texttt{assumes}\ \overline{\gamma \rightarrow \delta}\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \overline{lnk};\ \overline{md};\ \}$$
$$CT(\texttt{Object}) = \texttt{class}\ \texttt{Object<}\alpha_o\texttt{>}\ \{\ \}$$

Figure 3.3: FDJ auxiliary definitions. Adapted from (Aldrich and Chambers 2004), where *Aux-Domains* and *Aux-Fields* do not have base cases for `Object` (*Aux-Domains-Obj* and *Aux-Fields-Obj*).

$$\frac{S[\ell] = C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \qquad fields(C\texttt{<}\overline{p}\texttt{>}) = \overline{T}\ \overline{f}}{\ell.f_i;\ S \rightsquigarrow v_i;\ S}[\textit{R-Read}]$$

$$\frac{\ell \notin dom(S) \qquad S' = S[\ell \mapsto C\texttt{<}\overline{p}\texttt{>}(\overline{v})]}{\texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(\overline{v});\ S \rightsquigarrow \ell;\ S'}[\textit{R-New}]$$

$$\frac{S[\ell] = C\texttt{<}\overline{p}\texttt{>}(\overline{v}) \qquad mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x}, e_0)}{\ell.m(\overline{v});\ S \rightsquigarrow \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_0;\ S}[\textit{R-Invk}]$$

$$\frac{}{\ell \triangleright v;\ S \rightsquigarrow v;\ S}[\textit{R-Context}]$$

Figure 3.4: FDJ dynamic semantics. Source: (Aldrich and Chambers 2004).

$$\frac{S \vdash e_i \mapsto e_i', S'}{S \vdash \texttt{new}\ C\!<\!\overline{p}\!>\!(v_{1..i-1}, e_i, e_{i+1..n}) \mapsto \texttt{new}\ C\!<\!\overline{p}\!>\!(v_{1..i-1}, e_i', e_{i+1..n}), S'}\ \textit{RC-New}$$

$$\frac{S \vdash e \mapsto e', S'}{S \vdash e.f_i \mapsto e'.f_i, S'}\ \textit{RC-Read}$$

$$\frac{S \vdash e \mapsto e', S'}{S \vdash e.m(\overline{e}) \mapsto e'.m(\overline{e}), S'}\ \textit{RC-RecvInvk}$$

$$\frac{S \vdash e_i \mapsto e_i', S'}{S \vdash v.m(v_{1..i-1}, e_i, e_{i+1..n}) \mapsto v.m(v_{1..i-1}, e_i', e_{i+1..n}), S'}\ \textit{RC-ArgInvk}$$

$$\frac{S \vdash e \mapsto e', S'}{S \vdash \ell \triangleright e \mapsto \ell \triangleright e', S'}\ \textit{RC-Context}$$

Figure 3.5: FDJ congruence rules. Source: (Aldrich and Chambers 2004).

$$\frac{CT(C) = \texttt{class}\ C\!<\!\overline{\alpha, \beta}\!>\ \texttt{extends}\ C'\!<\!\overline{\alpha}\!>\ \ldots}{C\!<\!\overline{d, d'}\!>\ <:\ C'\!<\!\overline{d}\!>}\ \textit{Subtype-Class}$$

$$\frac{}{T <: T}\ \textit{Subtype-Reflex}$$

$$\frac{T <: T' \qquad T' <: T''}{T <: T''}\ \textit{Subtype-Trans}$$

Figure 3.6: FDJ subtyping rules. Source: (Aldrich and Chambers 2004).

$$\frac{\Gamma(x) = C<\overline{p}>}{\Gamma; \Sigma; n_{this} \vdash x : C<\overline{p}>} \textit{ T-Var}$$

$$\frac{\Sigma(\ell) = C<\overline{p}>}{\Gamma; \Sigma; n_{this} \vdash \ell : C<\overline{p}>} \textit{ T-Loc}$$

$$\frac{\begin{array}{c}\Gamma, \Sigma, n_{this} \models assumptions(C<\overline{p}>) \qquad \Gamma, \Sigma, n_{this} \vdash \overline{e} : \overline{T'} \\ fields(C<\overline{p}>) = \overline{T}\ \overline{f} \qquad \overline{T'} <: \overline{T} \qquad \Gamma, \Sigma, n_{this} \vdash n_{this} : T_{this} \\ owner(C<\overline{p}>) \in (domains(T_{this}) \cup owner(T_{this}))\end{array}}{\Gamma; \Sigma; n_{this} \vdash \texttt{new}\ C<\overline{p}>(\overline{e}) : C<\overline{p}>} \textit{ T-New}$$

$$\frac{\Gamma; \Sigma; n_{this} \vdash e_0 : T_0 \qquad fields(T_0)[e_0/\texttt{this}] = \overline{T}\ \overline{f}}{\Gamma; \Sigma; n_{this} \vdash e_0.f_i : T_i} \textit{ T-Read}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma; n_{this} \vdash e_0 : T_0 \qquad \Gamma; \Sigma; n_{this} \vdash \overline{e} : \overline{T_a} \\ mtype(m, T_0) = \overline{T} \to T_R \qquad \overline{T'} = \overline{T}[\overline{e}/\overline{x}, e_0/\texttt{this}] \qquad \overline{T_a} <: \overline{T'}\end{array}}{\Gamma; \Sigma; n_{this} \vdash e_0.m(\overline{e}) : T_R[\overline{e}/\overline{x}, e_0/\texttt{this}]} \textit{ T-Invk}$$

$$\frac{\Gamma; \Sigma; n_{this} \vdash e : T}{\Gamma; \Sigma; n_{this} \vdash \ell \triangleright e : T} \textit{ T-Context}$$

Figure 3.7: FDJ typing rules. Source: (Aldrich and Chambers 2004).

$$\frac{\begin{array}{c}\overline{md}\ OK\ in\ C \qquad fields(C'\!<\!\overline{\alpha}\!>) = \overline{T'}\ \overline{g} \qquad \overline{lnk}\ OK\ in\ C\!<\!\overline{\alpha}, \overline{\beta}\!> \\ \{\texttt{this} : C\!<\!\overline{\alpha}, \overline{\beta}\!>\};\ \emptyset;\ \texttt{this} \models \texttt{this} \rightarrow owner(\overline{T}) \\ K = C\!<\!\overline{\alpha}, \overline{\beta}\!>(\overline{T'}\ \overline{g},\ \overline{T}\ \overline{f})\ \{\ super(\overline{g});\ this.\overline{f} = \overline{f};\ \}\end{array}}{\texttt{class}\ C\!<\!\overline{\alpha}, \overline{\beta}\!>\ \texttt{extends}\ C'\!<\!\overline{\alpha}\!>\ \texttt{assumes}\ \overline{\gamma} \rightarrow \overline{\delta}\ \{\ \overline{T}\ \overline{f};\ K\ \overline{dom};\ \overline{lnk};\ \overline{md};\}\ OK}\ \textit{ClsOK}$$

$$\frac{\begin{array}{c}CT(C) = \texttt{class}\ C\!<\!\overline{\alpha}, \overline{\beta}\!>\ \texttt{extends}\ C'\!<\!\overline{\alpha}\!> \ldots \\ override(m, C'\!<\!\overline{\alpha}\!>, \overline{T} \rightarrow T_R) \\ \{\overline{x} : \overline{T};\ \texttt{this} : C\!<\!\overline{\alpha}, \overline{\beta}\!>\};\ \emptyset;\ \texttt{this} \vdash e : T_R \qquad T_R <: T \\ \{\overline{x} : \overline{T};\ \texttt{this} : C\!<\!\overline{\alpha}, \overline{\beta}\!>\};\ \emptyset;\ \texttt{this} \models \texttt{this} \rightarrow owner(\overline{T})\end{array}}{T_R\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ e;\ \}\ OK\ in\ C}\ \textit{MethOK}$$

$$\frac{\begin{array}{c}\{d_1, d_2\} \cap domains(C\!<\!\overline{\alpha}\!>) \neq \emptyset \\ d_1 \notin domains(C\!<\!\overline{\alpha}\!>) \Longrightarrow (\texttt{this} : C\!<\!\overline{\alpha}\!>; \emptyset; \texttt{this} \models d_1 \rightarrow owner(C\!<\!\overline{\alpha}\!>)) \\ d_2 \notin domains(C\!<\!\overline{\alpha}\!>) \Longrightarrow (\texttt{this} : C\!<\!\overline{\alpha}\!>; \emptyset; \texttt{this} \models \texttt{this} \rightarrow d_2)\end{array}}{\texttt{link}\ d_1 \rightarrow d_2\ OK\ in\ C\!<\!\overline{\alpha}\!>}\ \textit{LinkOK}$$

$$\frac{\forall \ell \in domain(\Sigma)\ \emptyset; \Sigma; \ell \models assumptions(\Sigma[\ell])}{\Sigma\ OK}\ \textit{T-Assumptions}$$

$$\frac{\begin{array}{c}domain(S) = domain(\Sigma) \qquad S[\ell] = C\!<\!\overline{\ell'.x}\!>(\overline{v}) \iff \Sigma[\ell] = C\!<\!\overline{\ell'.x}\!> \\ fields(\Sigma[\ell]) = \overline{T}\ \overline{f} \Longrightarrow (S[\ell, i] = \ell'') \wedge (\Sigma[\ell''] <: T_i) \qquad \Sigma\ OK \\ (S[\ell, i] = \ell'') \Longrightarrow (\emptyset, \Sigma, \ell \models \ell \rightarrow owner(\Sigma[\ell'']))\end{array}}{\Sigma \vdash S}\ \textit{T-Store}$$

Figure 3.8: FDJ class, method and store typing. Source: (Aldrich and Chambers 2004).

3.1.3 Ownership domain soundness

We restate some key results from the soundness of ownership domains (Aldrich and Chambers 2004).

Lemma 1 (Lemma)**.** *If* $mtype(m, D) = \overline{T} \to T_R$ *then* $mtype(m, C) = \overline{T} \to T_R$ *for all* $C <: D$.

Proof. By induction on the derivation of $C <: D$ and $mtype(m, D)$. □

Lemma 2 (Substitution Lemma)**.** *If* $\Gamma, \overline{x} : \overline{\tau} \vdash e : T$ *and* $\Gamma \vdash \overline{x'} : \overline{\tau'}$ *where* $\overline{\tau'} <: \overline{\tau}$*, then* $\Gamma \vdash [\overline{x'}/\overline{x}]e : T'$ *for some* $T' <: T$.

Proof. By induction on the typing rules. □

Lemma 3 (Weakening Lemma)**.** *If* $\Gamma \vdash e : C$*, then* $\Gamma, x : D \vdash e : C$.

Proof. By induction on the typing rules. □

Lemma 4 (Store Lemma)**.** *If* $fields(C\langle\overline{d}\rangle) = \overline{T}\ \overline{f}$ *and* $S[\ell] = C\langle\overline{d}\rangle(\overline{e})$ *and* $\overline{e} : \overline{T'}$ *then* $\overline{T'} <: \overline{T}$.

Proof. Based on the rules ***T-New*** and ***R-New***. □

Lemma 5 (Method Lemma)**.**

If $mtype(m, C\langle\overline{d}, \overline{d'}\rangle) = \overline{T} \to T_R$
and $mbody(m, C\langle\overline{d}, \overline{d'}\rangle) = (\overline{x}, e_R)$
then for some $D\langle\overline{d}\rangle$ with $C\langle\overline{d}, \overline{d'}\rangle <: D\langle\overline{d}\rangle$
there exists $T_0 <: T_R$ such that $\overline{x} : \overline{T}, \texttt{this} : D\langle\overline{d}\rangle \vdash e_R : T_0$

Proof. By induction on $mtype$.

□

Theorem 1 (FDJ Type Preservation, a.k.a. Subject Reduction)**.** *If* $\emptyset, \Sigma, n_{this} \vdash e : T$*,* $\Sigma \vdash S$*, and* $S \vdash e \mapsto e', S'$*, then there exists* $\Sigma' \supseteq \Sigma$ *and* $T' <: T$ *such that* $\emptyset, \Sigma', n_{this} \vdash e' : T'$ *and* $\Sigma' \vdash S'$.

Proof. By induction over the derivation of $S \vdash e \mapsto e', S'$, with a case analysis on the outermost reduction rule used. □

Theorem 2 (FDJ Progress)**.** *If* $\emptyset, \Sigma, n_{this} \vdash e : T$ *and* $\Sigma \vdash S$*—i.e., e is closed and well-typed, then either* e *is a value or else* $S \vdash e \mapsto e', S'$.

Proof. By induction over the derivation of $\emptyset, \Sigma, n_{this} \vdash e : T$. □

FDJ has additional properties, such as Link Soundness, which are discussed elsewhere (Aldrich and Chambers 2004).

3.2 Object Graph (OGraph)

An OGraph is a graph with two types of nodes, OObjects and ODomains. An OGraph also has edges, OEdges, between OObjects, that correspond to field points-to relations. We refer to an OObject, ODomain, and OEdge by the meta-variables O, D and E, respectively.

3.2.1 Data Types

The data type declarations for the OGraph are in Fig. 3.9. An OGraph G is the triplet $G = \langle PtO, PtD, PtE \rangle$. PtO is a set of OObjects. PtD maps a pair consisting of an OObject O and a local domain or a domain parameter d in the abstract syntax, i.e., (O, d), to an ODomain D. Effectively, PtD maintains a mapping from formal domain parameters to actual domains. PtE is a set of OEdges.

The analysis distinguishes between different instances of the same class C that are in different domains, even if created at the same `new` expression. In addition, the analysis treats an instance of class C with actual parameters $\overline{p}$ differently from another instance that has actual parameters $\overline{p'}$. Hence, the datatype of an OObject uses $C{<}\overline{D}{>}$ instead of just a type and an owning ODomain. Fig. 3.9 reflects this change, compared to the earlier data type declarations (Fig. 2.23). As in FDJ, an OObject's owning ODomain is the first element D_1 of $\overline{D}$. For the root OObject of an OGraph, the owning ODomain is D_{shared}, and the root type cannot have domain parameters. Thus, each OObject O represents all object allocations of type C in an ODomain D_1, that have domain parameters $D_2 \ldots D_n$, which represent some runtime domains.

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

To deal with recursive types, as we discussed in Section 2.4.2.3 (Page 54), an ODomain can have multiple parent OObjects, and not a single one, so an ODomain does not have an owning OObject in its representation.

Each OEdge E is a directed edge from a source OObject to a target OObject, and indicates the field reference `f`. Note that defining an OEdge from a source $(Domain, Type)$ pair to a target $(Domain, Type)$ would be less precise because that would not take into account the domain parameters associated with an OObject (the previous system adopted this definition (Abi-Antoun and Aldrich 2009a)). Effectively, we define an OEdge in terms of a source $(OwningDomain, Type, OtherDomainParams)$ triplet to a destination one.

In addition to the FDJ store S, we maintain the maps H and K (the instrumented operational semantics require those, as we discuss below). The map H maps each object location ℓ in the store to a unique OObject O. The map K maps each runtime domain represented as $\ell.d$ in the store to a unique ODomain D.

$$
\begin{array}{lll}
G \in \mathsf{OGraph} & ::= \langle \textbf{Objects} = PtO,\ \textbf{Domains} = PtD,\ \textbf{Edges} = PtE\ \rangle & \\
 & ::= \langle PtO, PtD, PtE \rangle & \\
D \in \mathsf{ODomain} & ::= \langle\ \textbf{Id} = D_{id}, \textbf{Domain} = C{::}d\ \rangle & \\
 & ::= \langle\ D_{id}, C{::}d\ \rangle & \\
O \in \mathsf{OObject} & ::= \langle\ \textbf{Id} = O_{id},\ \textbf{Type} = C{<}\overline{D}{>}\ \rangle & \\
 & ::= \langle\ O_{id}, C{<}\overline{D}{>}\ \rangle & \\
E \in \mathsf{OEdge} & ::= \langle\ \textbf{From} = O_{src}, \textbf{Field} = f,\ \textbf{To} = O_{dst}\ \rangle & \\
 & ::= \langle\ O_{src}, f,\ O_{dst}\ \rangle & \\
PtD & ::= \emptyset \mid PtD \cup \{\ (O, d) \mapsto D\ \} & \text{Points-to Domain} \\
PtO & ::= \emptyset \mid PtO \cup \{\ O\ \} & \text{Points-to Object} \\
PtE & ::= \emptyset \mid PtE \cup \{\ E\ \} & \text{Points-to Edge} \\
\Upsilon & ::= \emptyset \mid \Upsilon \cup \{\ C{<}\overline{D}{>}\ \} & \text{Visited objects} \\
H & ::= \emptyset \mid H \cup \{\ \ell \mapsto O\ \} & \text{Object map} \\
K & ::= \emptyset \mid K \cup \{\ \ell.d \mapsto D\ \} & \text{Domain map}
\end{array}
$$

Figure 3.9: Data type declarations for the OGraph.

3.2.2 Constraint-Based Specification

The analysis abstractly interprets the program, and maps concrete domain and field declarations in the program to abstract values in an OGraph, namely OObjects, ODomains, and OEdges.

Aliasing and subtyping. The analysis conservatively assumes that two objects of the same type in the same domain may alias. The rules use ownership domain subtyping (Rule *Subtype-Class* in Fig. 3.6), which follows standard nominal subtyping, and in addition, checks that all domain parameters are invariant with subtyping.

Judgement form. We use a constraint-based specification (Fig. 3.10) instead of transfer functions. This formalizes the static analysis as a set of inference rules, and makes it easier to prove soundness. The constraint system is solved by adding OObjects, ODomains and OEdges, as required, but unifying ODomains using a heuristic, for termination. The analysis of a program P is the least solution $G = \langle PtO, PtD, PtE \rangle$ of the following constraint system:

$$\emptyset, \emptyset, PtO, PtD, PtE \vdash P = (CT, e_{root})$$

The judgement form for expressions is as follows:

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

The O subscript on the turnstile captures the context-sensitivity. H is part of the instrumentation that maps locations to OObjects (Section 3.3.1). We omit H for most of the rules that do not

need it. The context Γ is the FDJ typing context. The context Υ tracks the list of the previously analyzed cases starting from the root expression, to avoid non-termination in the presence of recursive types. Υ records all the combinations of class and domain parameters that the analysis encounters in a call stack, starting from the root expression. Note that Υ is not the same as PtO because PtO is global, whereas Υ is specific to a call stack.

Rules. The interpretation starts with a program P consisting of a class table CT and a root expression, e_{root}. We require an OObject, O_{world}, which has a single ODomain, D_{shared}, corresponding to the global domain `shared`. For clarity, we qualify a domain d by the class that declares it, as $C{::}d$. Since the `shared` domain is global, we qualify it as ::`shared`. The OObject O_{world} does not correspond to an actual runtime object, but the analysis requires a dummy receiver for top-level code.

$$D_{shared} = \langle\, D_s,\ {::}\texttt{shared}\,\rangle$$
$$O_{world} = \langle\, O_{world},\ \texttt{Object<}D_{shared}\texttt{>}\,\rangle$$

The analysis starts out with the root expression e_{root} with an O_{world} context.

$$\emptyset, \emptyset, PtO, PtD, PtE \vdash_{O_{world}} e_{root}$$

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

Then, PT-NEW uses the auxiliary judgement PT-DOM to ensure that PtD has an ODomain corresponding to each domain that the class C locally declares. PT-DOM also processes the superclass, in order to include inherited domains[3].

PT-NEW then relies on the auxiliary judgement PT-FIELDS to ensure that PtE has an OEdge from O_C to each object in the target domain that is type compatible with the target type, using PT-LOOKUP. PT-FIELDS also processes the superclass, in order to include inherited fields.

PT-OBJ1 and PT-OBJ2 are the base cases for PT-DOM and PT-FIELDS, respectively, dealing with the root class, `Object`, and do not consult the superclass, to ensure that the derivation is finite. Recall, in FDJ, the class `Object` has no fields, domains, or methods.

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

PT-LOOKUP implements a similar subtyping relationship as the *Subtype-Class* rule in FDJ (Fig. 3.6). It compares both classes and that the actual ODomains are equal, by mapping the domains p_i into D_i using the current context O.

[3]In FDJ, *private* domains are misnamed, and really have a *protected* semantics (See *Aux-Domains* in Fig. 3.3).

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

$$CT(C) = \texttt{class}\ C\!<\!\overline{\alpha}, \overline{\beta}\!>\ \texttt{extends}\ C'\!<\!\overline{\alpha}\!>\ \ldots\ \{\ \overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}$$

$$\frac{\begin{array}{c}\forall(\texttt{domain}\ d_j) \in \overline{dom} \qquad D_j = \langle D_{id_j},\ d_j\,\rangle \qquad \{(O_C, d_j) \mapsto D_j\} \subseteq PtD \\ PtO, PtD, PtE \vdash_O ptdomains(C', O_C)\end{array}}{PtO, PtD, PtE \vdash_O ptdomains(C, O_C)}\ [\textsc{Pt-Dom}]$$

$$\frac{}{PtO, PtD, PtE \vdash_O ptdomains(\texttt{Object}, O_C)}\ [\textsc{Pt-Obj1}]$$

$$CT(\texttt{Object}) = \texttt{class}\ \texttt{Object}\!<\!\alpha_o\!>\ \{\ \} \qquad \frac{}{PtO, PtD, PtE \vdash_O ptfields(\texttt{Object}, O_C)}\ [\textsc{Pt-Obj2}]$$

$$\frac{\begin{array}{c}\forall(T_k\ f_k) \in \overline{T}\ \overline{f} \qquad owner(T_k) = p'_k \qquad D_k = PtD[(O_C, p'_k)] \\ \forall k\ PtO, PtD, PtE \vdash_{O_C} ptlookup(T_k) = O_k \quad \{\langle\, O_C,\ f_k,\ O_k\,\rangle\} \subseteq PtE \\ PtO, PtD, PtE \vdash_O ptfields(C', O_C)\end{array}}{PtO, PtD, PtE \vdash_O ptfields(C, O_C)}\ [\textsc{Pt-Fields}]$$

$$\frac{\begin{array}{c}O_k = \langle\, O_{id}, C\!<\!\overline{D}\!>\,\rangle \in PtO \qquad T' = C'\!<\!\overline{p'}\!> \qquad C <: C' \\ \forall i \in 1..|\overline{p'}| \qquad D'_i = PtD[(O, p'_i)] \qquad D'_i = D_i\end{array}}{PtO, PtD, PtE \vdash_O ptlookup(T') = O_k}\ [\textsc{Pt-Lookup}]$$

$$\frac{}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O x}\ [\textsc{Pt-Var}] \qquad \frac{}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O \ell}\ [\textsc{Pt-Loc}]$$

$$\frac{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0.f_k}\ [\textsc{Pt-Read}]$$

$$\frac{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0 \qquad \Gamma, \Upsilon, PtO, PtD, PtE \vdash_O \overline{e}}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0.m(\overline{e})}\ [\textsc{Pt-Invk}]$$

$$\frac{O_C = H[\ell] \qquad \Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O_C} e}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O,\,H} \ell \triangleright e}\ [\textsc{Pt-Context}]$$

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

Figure 3.10: Constraint-based specification of the object graph extraction analysis.

To make the induction go through, even though the points-to analysis only looks at the new expression, the analysis requires rules for all the expression types. The rule PT-NEW is the most interesting, and is the only one that modifies PtE.

The rules PT-VAR and PT-LOC for variables and locations, respectively, are uninteresting. In the case of PT-LOC, the store constraint PT-SIGMA enforces any necessary conditions on each location ℓ.

The field access and method invocation rules are more interesting. PT-READ analyzes the receiver of the field access. Similarly, PT-INVK analyzes the receiver and the actual arguments for the method invocation.

There are two other interesting rules. PT-CONTEXT analyzes method calls in progress $\ell \triangleright e$, where ℓ is the receiver, by moving into the context of the receiver object O_C. Finally, the induction requires an augmented store typing rule, PT-SIGMA, to ensure that method bodies have been analyzed for all objects in the store.

Recursion. The analysis must handle recursive types, which can lead an OGraph to grow arbitrarily deep. To get a finite OGraph and ensure that the analysis terminates, the analysis could stop expanding the object structure at a certain depth. However, merely truncating the recursion may lead to unsoundness, if it fails to reveal relations when child objects point to external objects, and the child objects are beyond the truncated depth.

Instead, the analysis creates a cycle in the OGraph when it reaches a similar context. The cycle creation happens when the same ODomain appears as the child of two OObjects. This justifies an ODomain not having an owning OObject.

In Section 3.4, we discuss how a DisplayGraph displays a potentially cyclic OGraph.

3.3 Object Graph Soundness

We demonstrate the object and edge soundness of an extracted object graph using a proof. The proof relies on an instrumentation of the FDJ dynamic semantics, an approximation relation, and standard Progress and Preservation theorems.

3.3.1 Instrumented Semantics

To prove the soundness of the analysis, we take the FDJ operational semantics (Fig. 3.4), and we instrument them (Fig. 3.11). This instrumentation is safe since discarding it produces exactly the previous semantics (Fig. 3.4). For instance, compare R-NEW to IR-NEW (the common parts of the rules are highlighted in Fig. 3.11). Also note that only IR-NEW requires an interesting instrumentation. The rules IR-READ, IR-INVK and IR-CONTEXT, again, are needed for the induction to go through, but do not impact the instrumentation.

The instrumented evaluation judgement form is as follows:

$$e; S; H; K \rightsquigarrow_G e'; S'; H'; K'$$

where $G = \langle PtO, PtD, PtE \rangle$ is the statically computed object graph.

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

$$\frac{\boxed{S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad fields(C\!<\!\overline{p}\!>) = \overline{T}\ \overline{f}}}{\boxed{\ell.f_i; S}; H; K \rightsquigarrow_G \boxed{v_i; S}; H; K}[\text{IR-READ}]$$

$$\frac{\boxed{S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad mbody(m, C\!<\!\overline{p}\!>) = (\overline{x}, e_R)}}{\boxed{\ell.m(\overline{v}); S}; H; K \rightsquigarrow_G \boxed{\ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R; S}; H; K}[\text{IR-INVK}]$$

$$\frac{}{\boxed{\ell \triangleright v; S}; H; K \rightsquigarrow_G \boxed{v;\ S}; H; K}[\text{IR-CONTEXT}]$$

Figure 3.11: Instrumented runtime semantics (core rules).

In IR-NEW (Fig. 3.11), the actual domains p_i passed to the class C being allocated are runtime domains, which K maps to static ODomains in PtD. We use H to lookup the OObject O_k for each value v_k passed to initialize the k^{th} field of the object being allocated, and ensure that the OEdge is in PtE.

The instrumented evaluation relation also includes congruence rules, similar to those in FDJ (Fig. 3.5), and which leave the instrumentation as is (Fig. 3.12).

$$\frac{e_i; S; H; K \rightsquigarrow_G e_i'; S'; H'; K'}{\texttt{new}\ C\!<\!\overline{p}\!>\!(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K \rightsquigarrow_G \texttt{new}\ C\!<\!\overline{p}\!>\!(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'}[\text{IRC-NEW}]$$

$$\frac{e_0; S; H; K \rightsquigarrow_G e_0'; S'; H'; K'}{e_0.f_i; S; H; K \rightsquigarrow_G e_0'.f_i; S'; H'; K'}[\text{IRC-READ}]$$

$$\frac{e_0; S; H; K \rightsquigarrow_G e_0'; S'; H'; K'}{e_0.m(\overline{e}); S; H; K \rightsquigarrow_G e_0'.m(\overline{e}); S'; H'; K'}[\text{IRC-RECVINVK}]$$

$$\frac{e_i; S; H; K \rightsquigarrow_G e_i'; S'; H'; K'}{v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K \rightsquigarrow_G v.m(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'}[\text{IRC-ARGINVK}]$$

$$\frac{e; S; H; K \rightsquigarrow_G e'; S'; H'; K'}{\ell \triangleright e; S; H; K \rightsquigarrow_G \ell \triangleright e'; S'; H'; K'}[\text{IRC-CONTEXT}]$$

Figure 3.12: Instrumented runtime semantics (congruence rules).

3.3.2 Approximation relation

We define an approximation relation $\sim$ between a state (S, H, K) and an analysis result (PtO, PtD, PtE) as follows:

Definition 7 (Approximation relation (PT-APPROX))**.**

$\forall\, \Sigma \vdash S, \quad (S, H, K) \sim (PtO, PtD, PtE)$

iff

$\forall \ell \in dom(S), \Sigma[\ell] = C\!<\!\overline{\ell'.d}\!>$

implies

$H[\ell] = O_C = \langle O_{id}, C\!<\!\overline{D}\!>\rangle \in PtO$

and $\forall \ell'_j.d_j \in \overline{\ell'.d} \quad K[\ell'_j.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(PtD)$

and $\forall d_i \in domains(C\!<\!\overline{\ell'.d}\!>) \qquad K[\ell.d_i] = D_i = \langle D_{id_i},\ d_i\,\rangle \qquad \{(O_C, d_i) \mapsto D_i\} \in PtD$

and $fields(\Sigma[\ell]) = \overline{T}\ \overline{f}$ *and* $\forall k, \forall \ell' \quad S[\ell, k] = \ell' \Longrightarrow E_k = \langle H[\ell], f_k, H[\ell']\rangle \in PtE$

3.3.3 Lemmas

The Progress and Preservation theorems require the following lemmas.

Lemma 6 (Points-to Substitution Lemma)**.**

If

$\Gamma, \overline{x} : \overline{\tau} \vdash e : T$

$\Gamma, \overline{x} : \overline{\tau}, \Upsilon, PtO, PtD, PtE \vdash_O e$

$\Gamma \vdash \overline{v} : \overline{\tau'}$ *where* $\overline{\tau'} <: [\overline{v}/\overline{x}]\overline{\tau}$

then

$\boxed{\Gamma \vdash [\overline{v}/\overline{x}]e : T' \text{ for some } T' <: [\overline{v}/\overline{x}]T}$

$\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O [\overline{v}/\overline{x}]e$

Proof. By induction on the $\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O$ relation. □

Lemma 7 (Points-to Weakening Lemma)**.**

$$\begin{array}{ll} \textit{If} & \Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e \\ \textit{then} & \Gamma, \Upsilon \cup \{C{<}\overline{D}{>}\}, PtO, PtD, PtE \vdash_O e \end{array}$$

Proof. By induction on the $\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O$ relation. □

Lemma 8 (Points-to Strengthening Lemma)**.**

$$\begin{array}{ll} \textit{If} & \Gamma, \emptyset, PtO, PtD, PtE \vdash_O \texttt{new}\, C{<}\overline{p}{>}(\overline{v}) \\ & \forall i \in 1..|\overline{p}| \qquad D_i = PtD[(O, p_i)] \\ & \Gamma, \Upsilon \cup \{C{<}\overline{D}{>}\}, PtO, PtD, PtE \vdash_{O'} e' \\ \textit{then} & \Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O'} e' \end{array}$$

Proof. By induction on the $\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O$ relation. We cover one interesting case.

Case PT-NEW: Then $e' = \texttt{new}\ C'{<}\overline{p'}{>}(\overline{e})$. There are several sub-cases to consider.

$$\frac{\begin{array}{c} \forall i \in 1..|\overline{p'}| \qquad D'_i = PtD[(O', p'_i)] \qquad params(C') = \overline{\alpha} \\ O_{C'} = \langle\, O_{id},\ C'{<}\overline{D'}{>}\, \rangle \qquad \{O_{C'}\} \subseteq PtO \qquad \{(O_{C'}, \alpha_i) \mapsto D'_i\} \subseteq PtD \\ PtO, PtD, PtE \vdash_{O'} ptdomains(C', O_{C'}) \\ PtO, PtD, PtE \vdash_{O'} ptfields(C', O_{C'}) \\ \forall m.\ mbody(m, C'{<}\overline{p'}{>}) = (\overline{x} : \overline{T},\ e_R) \\ C'{<}\overline{D'}{>} \notin \Upsilon \cup \{C{<}\overline{D}{>}\} \implies \\ \{\overline{x}{:}\overline{T},\ \texttt{this}{:}C'{<}\overline{p'}{>}\}, \Upsilon \cup \{C{<}\overline{D}{>}\} \cup \{C'{<}\overline{D'}{>}\}, PtO, PtD, PtE \vdash_{O_{C'}} e_R \\ \Gamma, \Upsilon \cup \{C{<}\overline{D}{>}\}, PtO, PtD, PtE \vdash_{O'} \overline{e} \end{array}}{\Gamma, \Upsilon \cup \{C{<}\overline{D}{>}\}, PtO, PtD, PtE \vdash_{O'} \texttt{new}\ C'{<}\overline{p'}{>}(\overline{e})}$$

Subcase $C'{<}\overline{D'}{>} \neq C{<}\overline{D}{>}$ and $C'{<}\overline{D'}{>} \notin \Upsilon \cup \{C{<}\overline{D}{>}\}$

$\{\overline{x}{:}\overline{T},\ \texttt{this}{:}C{<}\overline{p}{>}\}, \Upsilon \cup \{C{<}\overline{D}{>}\} \cup \{C'{<}\overline{D'}{>}\}, PtO, PtD, PtE \vdash_{O_{C'}} e_R$ By sub-derivation

$\{\overline{x}{:}\overline{T},\ \texttt{this}{:}C{<}\overline{p}{>}\}, \Upsilon \cup \{C'{<}\overline{D'}{>}\}, PtO, PtD, PtE \vdash_{O_{C'}} e_R$ By i.h.

$\Gamma, \Upsilon \cup \{C{<}\overline{D}{>}\}, PtO, PtD, PtE \vdash_{O'} \overline{e}$ By sub-derivation

$\Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O'} \overline{e}$ By i.h.

$\Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O'} e'$ By PT-NEW

Subcase $C'{<}\overline{D'}{>} \neq C{<}\overline{D}{>}$ and $C'{<}\overline{D'}{>} \in \Upsilon \cup \{C{<}\overline{D}{>}\}$

$\Gamma, \Upsilon \cup \{C{<}\overline{D}{>}\}, PtO, PtD, PtE \vdash_{O'} \overline{e}$ By sub-derivation

$\Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O'} \overline{e}$ By i.h.

$\Gamma, \Upsilon, PtO, PtD, PtE \vdash_{O'} e'$ By PT-NEW

Subcase $C'{<}\overline{D'}{>} = C{<}\overline{D}{>}$**,** i.e. $C'{<}\overline{D'}{>} \in \Upsilon \cup \{C{<}\overline{D}{>}\}$

$\{\overline{x}{:}\overline{T},\ \texttt{this}{:}C{<}\overline{p}{>}\}, \emptyset, PtO, PtD, PtE \vdash_{O_{C'}} e_R$ By inversion

$\{\overline{x}{:}\overline{T},\ \texttt{this}{:}C{<}\overline{p}{>}\}, \Upsilon \cup \{C{<}\overline{D}{>}\}, PtO, PtD, PtE \vdash_{O_{C'}} e_R$ By Points-to Weakening Lemma

□

Lemma 9 (Pt-Domains Lemma)**.**

$$
\begin{array}{ll}
\textit{If} & \emptyset, \Sigma, n_{this} \vdash e : T \\
& \Sigma \vdash S \\
& PtO, PtD, PtE \vdash_{CT,H} \Sigma \\
& PtO, PtD, PtE \vdash_O \texttt{new}\, C\!<\!\overline{p}\!>\!(\overline{v}) \\
& (S, H, K) \sim (PtO, PtD, PtE) \\
& PtO, PtD, PtE \vdash_O ptdomains(C, O_C) \\
& \forall i \in 1..|\overline{p}| \qquad D_i = PtD[(O, p_i)] \\
& O_C = \langle\, O_{id},\ C\!<\!\overline{D}\!>\, \rangle \qquad \{O_C\} \subseteq PtO \\
\textit{then} & \\
& \forall d_j \in domains(C\!<\!\overline{p}\!>) \quad D_j = PtD[(O_C, d_j)]
\end{array}
$$

Proof. By induction on the $PtO, PtD, PtE \vdash_O ptdomains(C, O_C)$ relation.
Case PT-DOM:.

$PtO, PtD, PtE \vdash_O \texttt{new}\ C\!<\!\overline{p}\!>\!(\overline{v})$	By assumption
$\forall i \in 1..\|\overline{p}\| \quad D_i = PtD[(O, p_i)]$	By sub-derivation of PT-NEW
$params(C) = \overline{\alpha}$	By sub-derivation of PT-NEW
$O_C = \langle\, O_{id},\ C\!<\!\overline{D}\!>\, \rangle$	By sub-derivation of PT-NEW
$\{O_C\} \subseteq PtO$	By sub-derivation of PT-NEW
$\{(O_C, \alpha_i) \mapsto D_i\} \subseteq PtD$	By sub-derivation of PT-NEW
$PtO, PtD, PtE \vdash_O ptdomains(C, O_C)$	By sub-derivation of PT-NEW
$\forall(\texttt{domain}\ d_j) \in \overline{dom} \qquad D_j = \langle D_{id_j},\ d_j\, \rangle$	By sub-derivation of PT-DOM
$\{(O_C, d_j) \mapsto D_j\} \subseteq PtD$	By sub-derivation of PT-DOM
$\overline{dom} \in domains(C\!<\!\overline{p}\!>)$	By definition of $domains$
$PtO, PtD, PtE \vdash_O ptdomains(C', O_C)$	By sub-derivation of PT-DOM

Subcase $C' \neq \texttt{Object}$

By i.h.

Subcase $C' = \texttt{Object}$

$ptdomains(C', O_C) = \emptyset$ By definition of *Aux-Domains-Obj*

Case PT-OBJ1: Is immediate.

□

Lemma 10 (Pt-Fields Lemma)**.**

If $\emptyset, \Sigma, n_{this} \vdash e : T$
$\Sigma \vdash S$
$PtO, PtD, PtE \vdash_{CT,H} \Sigma$
$PtO, PtD, PtE \vdash_O \texttt{new}\, C{<}\overline{p}{>}(\overline{v})$
$(S, H, K) \sim (PtO, PtD, PtE)$
$PtO, PtD, PtE \vdash_O ptfields(C, O_C)$
$\forall i \in 1..|\overline{p}| \qquad D_i = PtD[(O, p_i)]$
$O_C = \langle\, O_{id},\ C{<}\overline{D}{>}\, \rangle \qquad \{O_C\} \subseteq PtO$

then

(1) $\quad O_k = H[v_k]$

(2) $\quad \forall (T_k\ f_k) \in fields(C{<}\overline{p}{>}) \quad E_k = \langle O_C, f_k, O_k \rangle \qquad E_k \in PtE$

Proof. By induction on the $PtO, PtD, PtE \vdash_O ptfields(C, O_C)$ relation.
Case PT-FIELDS:.

$PtO, PtD, PtE \vdash_O \texttt{new}\ C{<}\overline{p}{>}(\overline{v})$	By assumption
$PtO, PtD, PtE \vdash_O ptfields(C, O_C)$	By sub-derivation of PT-NEW
$\forall (T_k\ f_k) \in \overline{T}\ \overline{f}$	By sub-derivation of PT-FIELDS
$\quad owner(T_k) = p'_k$	By sub-derivation of PT-FIELDS
$\quad D_k = PtD[(O_C, p'_k)]$	By sub-derivation of PT-FIELDS
$\forall k\ PtO, PtD, PtE \vdash_{O_C} ptlookup(T_k) = O_k$	By sub-derivation of PT-FIELDS
$\{\langle\, O_C,\ f_k,\ O_k\, \rangle\} \subseteq PtE$	By sub-derivation of PT-FIELDS
$O_k = \langle\, O_{id}, C_k{<}\overline{D_k}{>}\, \rangle \in PtO$	By inversion of PT-LOOKUP
$\quad T_k = C'_k{<}\overline{p'}{>} \qquad C_k <: C'_k$	By inversion of PT-LOOKUP
$\forall i \in 1..\lvert\overline{p'}\rvert \qquad D_{k'_i} = PtD[(O_C, p'_i)] \qquad D_{k'_i} = D_{k_i}$	By inversion of PT-LOOKUP
$PtO, PtD, PtE \vdash_O ptfields(C', O_C)$	By sub-derivation of PT-FIELDS

Subcase $C' \neq \texttt{Object}$

By i.h.

Subcase $C' = \texttt{Object}$

$ptfields(C', O_C) = \emptyset$ By definition of *Aux-Fields-Obj*

Case PT-OBJ2:. Is immediate.
This shows (2).

To show (1), we use the approximation relation PT-APPROX.

$T_k\ f_k \in \overline{T}\ \overline{f}$ By sub-derivation of PT-FIELDS

$(S, H, K) \sim (PtO, PtD, PtE)$ By assumption

$\forall v_k \in dom(S), \Sigma[v_k] = T_k{<}\overline{v'_k.d}{>}$ By PT-APPROX

implies

$H[v_k] = O_k = \langle O_{id}, T_k{<}\overline{D}{>}\rangle \in PtO$

and $\forall v_{k'_j}.d_j \in \overline{v'_k.d}$

$K[v_{k'_j}.d_j] = D_j = \langle D_{id_j}, d_j\rangle \in rng(PtD)$

and $\forall d_i \in domains(T_k{<}\overline{v'_k.d}{>})$

$K[v_k.d_i] = D_i = \langle D_{id_i},\ d_i\ \rangle \qquad \{(O_k, d_i) \mapsto D_i\} \in PtD$

and $fields(\Sigma[v_k]) = \overline{T}\ \overline{f}$

and $\forall k, \forall v'_k \quad S[v_k, k] = v'_k \Longrightarrow E_k = \langle H[v_k], f_k, H[v'_k]\rangle \in PtE$

□

3.3.4 Preservation

Theorem 3 (Points-to Preservation (Subject Reduction))**.**

If

$\boxed{\emptyset, \Sigma, n_{this} \vdash e : T}$

$\boxed{\Sigma \vdash S}$

$PtO, PtD, PtE \vdash_{CT,H} \Sigma$

$PtO, PtD, PtE \vdash_{O} e$

$(S, H, K) \sim (PtO, PtD, PtE)$

$\boxed{e; S}; H; K \rightsquigarrow_{G} \boxed{e'; S'}; H'; K'$

then

$\boxed{\textit{there exists } \Sigma' \supseteq \Sigma \textit{ and } T' <: T \textit{ such that } \emptyset, \Sigma', n_{this} \vdash e' : T' \textit{ and } \Sigma' \vdash S'}$,

$(S', H', K') \sim (PtO, PtD, PtE)$,

$PtO, PtD, PtE \vdash_{O} e'$,

and $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

The Points-to Subject Reduction theorem extends the FDJ Subject Reduction (the common parts are highlighted). Those parts are proved by induction over the derivation of the FDJ evaluation relation $e;\ S \rightsquigarrow e';\ S'$ (Fig. 3.4).

Proof. We prove Points-To Preservation by induction on the instrumented evaluation relation $e; S; H; K \rightsquigarrow_{G} e'; S'; H'; K'$ with a case analysis on the outermost reduction rule used.

Case IR-NEW: Then $e = \texttt{new}\ C{<}\overline{\ell'.d}{>}(\overline{v})$. And $e' = \ell$.

To show:
(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$PtO, PtD, PtE \vdash_O e$ and $(S, H, K) \sim (PtO, PtD, PtE)$ — By assumption
$\forall \iota \in dom(S), \Sigma[\iota] = C{<}\overline{\iota'.d}{>}$ — Since $\Sigma \vdash S$
$\implies$ — By PT-APPROX
$H[\iota] = O_\iota = \langle O_{id}, C{<}\overline{D}{>}\rangle \in PtO$ — By PT-APPROX
and $\forall \iota'_j.d_j \in \overline{\iota'.d} \quad K[\iota'_j.d_j] = D_{\iota j} = \langle D_{id_j}, d_j\rangle \in rng(PtD)$ — By PT-APPROX
and $\forall d_j \in domains(C{<}\overline{p}{>})$ — By PT-APPROX
$K[\iota.d_j] = D_{\iota j} = \langle D_{id_j},\ d_j\ \rangle \qquad \{(O_\iota, d_j) \mapsto D_{\iota j}\} \in PtD$ — By PT-APPROX
and $fields(\Sigma[\iota]) = \overline{T}\ \overline{f}$ — By PT-APPROX
and $\forall k, \forall \iota' \quad S[\iota, k] = \iota' \implies \langle H[\iota], f_k, H[\iota']\rangle \in PtE$ — By PT-APPROX

We also have:

$O_C = \langle O_{id}, C{<}\overline{D}{>}\rangle \in PtO$ — By sub-derivation of IR-NEW
$S' = S[\ell \mapsto C{<}\overline{\ell'.d}{>}(\overline{v})]$ — By sub-derivation of IR-NEW
$H' = H[\ell \mapsto O_C]$ — By sub-derivation of IR-NEW
$\forall i \in |\ell'.d| \quad D_i = K[\ell'_i.d_i]$ — By sub-derivation of IR-NEW
$\forall d_j \in domains(C{<}\overline{\ell'.d}{>})$ — By sub-derivation of IR-NEW
$D_j = PtD[(O_C, d_j)]$ — By sub-derivation of IR-NEW
$K' = K[\ell.d_j \mapsto D_j]$ — By sub-derivation of IR-NEW
$\exists \Sigma' \supseteq \Sigma$ s.t. $\Sigma'[\ell] = C{<}\overline{\ell'.d}{>}$
$\forall T_k f_k \in fields(\Sigma'[\ell])$ s.t. $S[\ell, k] = v_k$ — By sub-derivation of IR-NEW
$O_k = H[v_k]$ — By sub-derivation of IR-NEW
$E_k = \langle O_C, f_k, O_k\rangle \in PtE$ — By sub-derivation of IR-NEW
$(S', H', K') \sim (PtO, PtD, PtE)$ — By PT-APPROX
This proves (1)

$PtO, PtD, PtE \vdash_O e'$ — By PT-LOC since $e' = \ell$
This proves (2)

$PtO, PtD, PtE \vdash_{CT,H} \Sigma$ By assumption

$\forall \iota \in dom(S), \Sigma[\iota] = C_\iota\texttt{<}\overline{p}\texttt{>}$ By sub-derivation of PT-SIGMA

$\quad H[\iota] = O_\iota = \langle O_{id}, C_\iota\texttt{<}\overline{D}\texttt{>}\rangle \in PtO$

$\quad \forall m.\ mbody(m, C_\iota\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R)$

$\quad \{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \emptyset, PtO, PtD, PtE \vdash_{O_\iota} e_R$

$O_C = \langle O_{id}, C\texttt{<}\overline{D}\texttt{>}\rangle \in PtO$ By sub-derivation of IR-NEW

$S' = S[\ell \mapsto C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v})]$ By sub-derivation of IR-NEW

$H' = H[\ell \mapsto O_C]$ By sub-derivation of IR-NEW

$PtO, PtD, PtE \vdash_O e$ By assumption with e below

$e = \texttt{new}\ C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v})$

$\forall m.\ mbody(m, C\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R) \qquad C\texttt{<}\overline{D}\texttt{>} \notin \Upsilon \implies$ By sub-derivation of PT-NEW

$\{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \Upsilon \cup \{C\texttt{<}\overline{D}\texttt{>}\}, PtO, PtD, PtE \vdash_{O_C} e_R$ Since $\Upsilon = \emptyset$

$\{\overline{x} : \overline{T},\ \texttt{this} : C\texttt{<}\overline{p}\texttt{>}\}, \emptyset, PtO, PtD, PtE \vdash_{O_C} e_R$ By Points-to Strengthening Lemma

$\forall \iota \in dom(S'), \Sigma'[\iota] = C\texttt{<}\overline{p}\texttt{>}$ By above

$\quad H'[\iota] = O_\iota = \langle O_{id}, C_\iota\texttt{<}\overline{D}\texttt{>}\rangle \in PtO$

$\quad \forall m.\ mbody(m, C_\iota\texttt{<}\overline{p}\texttt{>}) = (\overline{x} : \overline{T},\ e_R)$

$\quad \{\overline{x} : \overline{T},\ \texttt{this} : C_\iota\texttt{<}\overline{p}\texttt{>}\}, \emptyset, PtO, PtD, PtE \vdash_{O_\iota} e_R$

$PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$ By PT-SIGMA with above H' and Σ'

This proves (3)

Case IR-READ: Then $e = \ell.f_k$. And $e' = v_k$. To show:

(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$(S, H, K) \sim (PtO, PtD, PtE)$ — By assumption
$S' = S, H' = H, K' = K$ — By sub-derivation of IR-READ
This proves (1)

$PtO, PtD, PtE \vdash_O e'$ — By PT-LOC since $e' = v_k$
This proves (2)

$PtO, PtD, PtE \vdash_{CT,H} \Sigma$ — By PT-SIGMA
$S' = S, H' = H, K' = K$ — By sub-derivation of IR-READ
This proves (3) — Take $\Sigma' = \Sigma$

Case IR-CONTEXT: Then $e = \ell \triangleright v$. And $e' = v$. To show:

(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$(S, H, K) \sim (PtO, PtD, PtE)$ — By assumption
$S' = S, H' = H, K' = K$ — By sub-derivation of IR-CONTEXT
This proves (1)

$PtO, PtD, PtE \vdash_O e'$ — By PT-LOC since $e' = v$
This proves (2)

$PtO, PtD, PtE \vdash_{CT,H} \Sigma$ — By PT-SIGMA
$S' = S, H' = H, K' = K$ — By sub-derivation of IR-CONTEXT
This proves (3) — Take $\Sigma' = \Sigma$

Case IR-INVK: Then $e = \ell.m(\overline{v})$. And $e' = \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$.

To show:
(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$(S, H, K) \sim (PtO, PtD, PtE)$ By assumption
$S' = S, H' = H, K' = K$ By sub-derivation of IR-INVK
This proves (1)

From PT-INVK:

$$\frac{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0 \qquad \Gamma, \Upsilon, PtO, PtD, PtE \vdash_O \overline{e}}{\Gamma, \Upsilon, PtO, PtD, PtE \vdash_O e_0.m(\overline{e})}$$

From *MethOK*:

$$mtype(m, T_0) = \overline{T} \to T_R \qquad \{\overline{x} : \overline{T},\ \texttt{this} : C{<}\overline{\alpha}, \overline{\beta}{>}\},\ \emptyset,\ \texttt{this} \vdash e_R : T_R \qquad T_R <: T$$

$S[\ell] = C\langle\overline{d}, \overline{d'}\rangle(\overline{v})$ By sub-derivation of IR-INVK

$mbody(m, C\langle\overline{d}, \overline{d'}\rangle) = (\overline{x}, e_R)$ By sub-derivation of IR-INVK

$e_0 = \ell$

$\Sigma[\ell] = C\langle\overline{d}, \overline{d'}\rangle = T_0$ T-Store

$e_0 : C\langle\overline{d}, \overline{d'}\rangle$

$mtype(m, C\langle\overline{d}, \overline{d'}\rangle) = \overline{T} \to T_R$

$\overline{v} : \overline{T_a}$ By inversion

$\overline{T_a} <: [\overline{v}/\overline{x}, \ell/\texttt{this}]\,\overline{T}$ for some $\overline{T_a}$ and $\overline{T}$

There are some $D\langle\overline{d}\rangle$ and T_0 s.t. By Method Lemma (page 83)

$T_0 <: T_R$ and $C\langle\overline{d}, \overline{d'}\rangle <: D\langle\overline{d}\rangle$

s.t. $\{\overline{x} : \overline{T}, \texttt{this} : D\langle\overline{d}\rangle\} \vdash e_R : T_R$

$\Gamma, \{\overline{x} : \overline{T},\ \texttt{this} : C\langle\overline{d}, \overline{d'}\rangle\}, \emptyset, PtO, PtD, PtE \vdash_{O_C} e_R$ By PT-SIGMA

$O_C = H[\ell]$ By PT-SIGMA

Since term substitution preserves typing, there exists some T_S

$T_S <: C\langle\overline{d}, \overline{d'}\rangle$ such that $[\overline{v}/\overline{x}, \ell/\texttt{this}]e_R : T_S$

$T_S <: T_0$ and $T_0 <: T_R$ By above

$T_S <: T_R$ By transitivity of $<:$

Take $T = T' = T_R$ Preservation

$PtO, PtD, PtE \vdash_O \ell$ By PT-LOC

$\Gamma, \emptyset, PtO, PtD, PtE \vdash_{O_C} [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$ By Points-to Substitution Lemma

$\Gamma, \emptyset, PtO, PtD, PtE \vdash_O \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$ By PT-CONTEXT

This proves (2)

$PtO, PtD, PtE \vdash_{CT,H} \Sigma$ By PT-SIGMA

$S' = S, H' = H, K' = K$ By sub-derivation of IR-CONTEXT

This proves (3) Take $\Sigma' = \Sigma$

Case IRC-READ: Then $e = e_0.f_k$. And $e' = e'_0.f_k$. To show:

(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-READ
$(S', H', K') \sim (PtO, PtD, PtE)$	By induction hypothesis
This proves (1)	

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-READ
$PtO, PtD, PtE \vdash_O e'_0$	By induction hypothesis
$PtO, PtD, PtE \vdash_O e'_0.f_k$	By PT-READ
This proves (2)	

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-READ
$PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$	By induction hypothesis
This proves (3)	Take $\Sigma' = \Sigma$

Case IRC-RECVINVK: Then $e = e_0.m(\overline{e})$. And $e' = e'_0.m(\overline{e})$. To show:

(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-RECVINVK
$(S', H', K') \sim (PtO, PtD, PtE)$	By induction hypothesis
This proves (1)	

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-RECVINVK
$PtO, PtD, PtE \vdash_O e'_0$	By induction hypothesis
$PtO, PtD, PtE \vdash_O \overline{e}$	By PT-INVK
$PtO, PtD, PtE \vdash_O e'_0.m(\overline{e})$	By PT-INVK
This proves (2)	

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-RECVINVK
$PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$	By induction hypothesis
This proves (3)	Take $\Sigma' = \Sigma$

Case IRC-ARGINVK: Then $e = v.m(v_{1..i-1}, e_i, e_{i+1..n})$. And $e' = v.m(v_{1..i-1}, e'_i, e_{i+1..n})$. To show:

(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$e_i; S; H; K \rightsquigarrow_G e'_i; S'; H'; K'$	By sub-derivation of IRC-ARGINVK
$(S', H', K') \sim (PtO, PtD, PtE)$	By induction hypothesis
This proves (1)	

$e_i; S; H; K \rightsquigarrow_G e'_i; S'; H'; K'$	By sub-derivation of IRC-ARGINVK
$PtO, PtD, PtE \vdash_O e'_i$	By induction hypothesis
$PtO, PtD, PtE \vdash_O v.m(v_{1..i-1}, e'_i, e_{i+1..n})$	By PT-INVK
This proves (2)	

$e_i; S; H; K \rightsquigarrow_G e'_i; S'; H'; K'$	By sub-derivation of IRC-ARGINVK
$PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$	By induction hypothesis
This proves (3)	Take $\Sigma' = \Sigma$

Case IRC-NEW: Then $e = \texttt{new}\ C{<}\overline{p}{>}(v_{1..i-1}, e_i, e_{i+1..n})$. And $e' = \texttt{new}\ C{<}\overline{p}{>}(v_{1..i-1}, e'_i, e_{i+1..n})$. To show:

(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$e_i; S; H; K \rightsquigarrow_G e'_i; S'; H'; K'$	By sub-derivation of IRC-NEW
$(S', H', K') \sim (PtO, PtD, PtE)$	By induction hypothesis
This proves (1)	

$e_i; S; H; K \rightsquigarrow_G e'_i; S'; H'; K'$	By sub-derivation of IRC-NEW
$PtO, PtD, PtE \vdash_O e'_i$	By induction hypothesis
$PtO, PtD, PtE \vdash_O \texttt{new}\ C{<}\overline{p}{>}(v_{1..i-1}, e'_i, e_{i+1..n})$	By PT-NEW
This proves (2)	

$e_i; S; H; K \rightsquigarrow_G e'_i; S'; H'; K'$	By sub-derivation of IRC-NEW
$PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$	By induction hypothesis
This proves (3)	Take $\Sigma' = \Sigma$

Case IRC-CONTEXT: Then $e = \ell \triangleright e_0$. And $e' = \ell \triangleright e'_0$. To show:

(1) $(S', H', K') \sim (PtO, PtD, PtE)$
(2) $PtO, PtD, PtE \vdash_O e'$
(3) $PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-CONTEXT
$(S', H', K') \sim (PtO, PtD, PtE)$	By induction hypothesis
This proves (1)	

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-CONTEXT
$PtO, PtD, PtE \vdash_O e'_0$	By induction hypothesis
$PtO, PtD, PtE \vdash_O \ell \triangleright e'_0$	By PT-CONTEXT
This proves (2)	

$e_0; S; H; K \rightsquigarrow_G e'_0; S'; H'; K'$	By sub-derivation of IRC-CONTEXT
$PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$	By induction hypothesis
This proves (3)	Take $\Sigma' = \Sigma$

□

Because we added instrumentation to the runtime semantics, we also need to prove progress, i.e., the instrumentation will not cause the program to get stuck during evaluation.

3.3.5 Progress

Theorem 4 (Points-to Progress)**.**

If

$\boxed{\emptyset, \Sigma, n_{this} \vdash e : T}$

$\boxed{\Sigma \vdash S}$

$PtO, PtD, PtE \vdash_{CT,H} \Sigma$

$PtO, PtD, PtE \vdash_{O} e$

$(S, H, K) \sim (PtO, PtD, PtE)$

then

either $\boxed{e \text{ is a value}}$

or else $\boxed{e; S}; H; K \rightsquigarrow_G \boxed{e'; S'}; H'; K'$

Proof. We prove Points-to Progress by induction over the derivation of $PtO, PtD, PtE \vdash_{O} e$, with a case analysis on the last typing rule used.

Case PT-NEW: Then there are two sub-cases to consider, depending on whether $\overline{e}$ are values.
Subcase $e = \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n})$. Then IRC-NEW can apply.

IRC-NEW

$$\text{From}\ \frac{e_i; S; H; K \rightsquigarrow_G e_i'; S'; H'; K'}{\texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K \rightsquigarrow_G \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'}$$

$PtO, PtD, PtE \vdash_O e_i$ — By sub-derivation of PT-NEW

$e_i; S; H; K \rightsquigarrow_G e_i'; S'; H'; K'$ — By induction hypothesis

$\texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K \rightsquigarrow_G$
$\texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i', e_{i+1..n}); S'; H; K$ — By IRC-NEW

Take $e' = \texttt{new}\ C\texttt{<}\overline{p}\texttt{>}(v_{1..i-1}, e_i', e_{i+1..n})$

Subcase $e = \texttt{new}\ C\texttt{<}\overline{\ell'.d}\texttt{>}(\overline{v})$. Take $e' = \ell$. Then IR-NEW can apply.

IR-NEW

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

To show:

(1) $\forall i \in |\overline{\ell'.d}| \quad D_i = K[\ell_i'.d_i]$
(2) $O_C = \langle O_{id}, C\texttt{<}\overline{D}\texttt{>}\rangle \qquad O_C \in PtO$
(3) $\forall d_j \in domains(C\texttt{<}\overline{\ell'.d}\texttt{>}) \quad D_j = PtD[(O_C, d_j)]$
(4) $O_k = H[v_k]$
(5) $\forall k \in fields(C\texttt{<}\overline{\ell'.d}\texttt{>}) \quad E_k = \langle O_C, f_k, O_k\rangle \qquad E_k \in PtE$

$(S, H, K) \sim (PtO, PtD, PtE)$ — By assumption

$\forall \iota \in dom(S), \Sigma[\iota] = C_\iota\texttt{<}\overline{\iota'.d}\texttt{>}$ — Since $\Sigma \vdash S$

$H[\iota] = O_\iota = \langle O_{id}, C_\iota\texttt{<}\overline{D}\texttt{>}\rangle \in PtO$ — By PT-APPROX

and $\forall \iota_j'.d_j \in \overline{\iota'.d} \quad K[\iota_j'.d_j] = D_{\iota j} = \langle D_{id_j}, d_j\rangle \in rng(PtD)$ — By PT-APPROX

and $\forall d_{\iota j} \in domains(C_\iota\texttt{<}\overline{\iota'.d}\texttt{>})$ — By PT-APPROX

$K[\iota.d_j] = D_{\iota j} = \langle D_{id_{\iota j}},\ d_{\iota j}\ \rangle \qquad \{(O_\iota, d_{\iota j}) \mapsto D_{\iota j}\} \in PtD$ — By PT-APPROX

and $fields(\Sigma[\iota]) = \overline{T}\ \overline{f}$ — By PT-APPROX

and $\forall k, \forall \iota' \quad S[\iota, k] = \iota' \Longrightarrow E_k = \langle H[\iota], f_k, H[\iota']\rangle \in PtE$ — By PT-APPROX

This proves (1)

$PtO, PtD, PtE \vdash_O e$ — By assumption

$\forall i \in 1..|\ell'.d| \quad D_i = PtD[(O, p_i)]$ — By sub-derivation of PT-NEW

$params(C) = \overline{\alpha}$ — By sub-derivation of PT-NEW

$O_C = \langle\, O_{id},\ C\!<\!\overline{D}\!> \,\rangle$ — By sub-derivation of PT-NEW

$\{O_C\} \subseteq PtO$ — By sub-derivation of PT-NEW

This proves (2)

$CT(C) = \texttt{class}\ C\!<\!\overline{\alpha}, \overline{\beta}\!>\ \texttt{extends}\ C'\!<\!\overline{\alpha}\!>\ \ldots\ \{$
$\overline{T}\ \overline{f};\ \overline{dom};\ \ldots;\ \overline{md};\ \}$

$\{(O_C, \alpha_i) \mapsto D_i\} \subseteq PtD$ — By sub-derivation of PT-NEW

$PtO, PtD, PtE \vdash_O ptdomains(C, O_C)$ — By sub-derivation of PT-NEW

This proves (3) — By Pt-Domains Lemma

$PtO, PtD, PtE \vdash_{CT,H} \Sigma$ — By assumption

$\forall \iota \in dom(S), \Sigma[\iota] = C_\iota\!<\!\overline{p}\!>$ — By sub-derivation of PT-SIGMA

$H[\iota] = O_\iota = \langle O_{id}, C_\iota\!<\!\overline{D}\!>\rangle \in PtO$ — By sub-derivation of PT-SIGMA

This proves (4)

$PtO, PtD, PtE \vdash_O ptfields(C, O_C)$ — By sub-derivation of PT-NEW

This proves (5) — By Pt-Fields Lemma

Case PT-VAR: Then $e = x$.

Not applicable since variable is not a closed term.

Case PT-LOC: Then $e = \ell$.

e is value.

Case PT-READ: Then $e = e_0.f_k$. There are two sub-cases to consider, depending on whether the receiver e_0 is a value.

Subcase $e_0 = \ell$. Then $e = \ell.f_i$.

From $\dfrac{\text{IR-READ}\quad S[\ell] = C{<}\overline{p}{>}(\overline{v}) \qquad fields(C{<}\overline{p}{>}) = \overline{T}\ \overline{f}}{\ell.f_i; S; H; K \rightsquigarrow_G v_i; S; H; K}$

Take $e' = v_i$

Then IR-READ can apply. By ordinary FDJ progress.

Subcase $e_0 = e_0'.f_i$.

From $\dfrac{\text{IRC-READ}\quad e_0; S; H; K \rightsquigarrow_G e_0'; S'; H'; K'}{e_0.f_i; S; H; K \rightsquigarrow_G e_0'.f_i; S'; H'; K'}$

$e_0'; S; H; K \rightsquigarrow_G e_0''; S'; H'; K'$ By induction hypothesis

$e_0'.f_i; S; H; K \rightsquigarrow_G e_0''.f_i; S''; H''; K''$ By IRC-READ

Take $e' = e_0''.f_i$

Case PT-INVK: Then $e = e_0.m(\overline{e})$. There are three sub-cases to consider, depending on whether the receiver e_0, or the method arguments are values.

Subcase $e_0 = \ell$, and $\overline{e} = \overline{v}$, that is, $e = \ell.m(\overline{v})$.

From $\dfrac{\text{IR-INVK}\quad S[\ell] = C\!<\!\overline{p}\!>\!(\overline{v}) \qquad mbody(m, C\!<\!\overline{p}\!>) = (\overline{x}, e_R)}{\ell.m(\overline{v}); S; H; K \rightsquigarrow_G \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R; S; H; K}$

Then IR-INVK can apply.	By ordinary FDJ progress.
Take $e' = \ell \triangleright [\overline{v}/\overline{x}, \ell/\texttt{this}]e_R$	

Subcase $e_0 = e_0'$, that is $e = e_0'.m(\overline{e})$.

From $\dfrac{\text{IRC-RECVINVK}\quad e_0; S; H; K \rightsquigarrow_G e_0'; S'; H'; K'}{e_0.m(\overline{e}); S; H; K \rightsquigarrow_G e_0'.m(\overline{e}); S'; H'; K'}$

$e_0'; S; H; K \rightsquigarrow_G e_0''; S'; H'; K'$	By induction hypothesis
$e_0'.m(\overline{e}); S; H; K \rightsquigarrow_G e_0''.m(\overline{e}); S''; H''; K''$	By IRC-RECVINVK
Take $e' = e_0''.m(\overline{e})$	

Subcase $e_0 = v$, that is, $e = v.m(v_{1..i-1}, e_i, e_{i+1..n})$.

From $\dfrac{\text{IRC-ARGINVK}\quad e_i; S; H; K \rightsquigarrow_G e_i'; S'; H'; K'}{v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K \rightsquigarrow_G v.m(v_{1..i-1}, e_i', e_{i+1..n}); S'; H'; K'}$

$PtO, PtD, PtE \vdash_O e_i$	By sub-derivation of PT-INVK
$e_i; S; H; K \rightsquigarrow_G e_i'; S'; H'; K'$	By induction hypothesis
$v.m(v_{1..i-1}, e_i, e_{i+1..n}); S; H; K \rightsquigarrow_G v.m(v_{1..i-1}, e_i', e_{i+1..n}); S''; H''; K''$	By IRC-ARGINVK
Take $e' = v.m(v_{1..i-1}, e_i', e_{i+1..n})$	

Case PT-CONTEXT: Then $e = \ell \triangleright e_0$. There are two sub-cases to consider, depending on whether e_0 is a value.

Subcase e_0 is a value, i.e., $e = \ell \triangleright v$.

From $\dfrac{}{\ell \triangleright v; S; H; K \rightsquigarrow_G v; S; H; K}$ IR-CONTEXT

Then IR-CONTEXT can apply

Take $e' = v$

Subcase e_0 is not a value.

From $\dfrac{e_0; S; H; K \rightsquigarrow_G e_0'; S'; H'; K'}{\ell \triangleright e_0; S; H; K \rightsquigarrow_G \ell \triangleright e_0'; S'; H'; K'}$ IRC-CONTEXT

$e_0; S; H; K \rightsquigarrow_G e_0'; S'; H'; K'$	By induction hypothesis
$\ell \triangleright e_0'; S; H; K \rightsquigarrow_G \ell \triangleright e_0'; S'; H'; K'$	By IRC-CONTEXT
Take $e' = \ell \triangleright e_0'$	

□

3.3.6 Object Graph Soundness

An OGraph is a *sound* approximation of a Runtime Object Graph (ROG) represented by a well-typed store S, for any program run, when the OGraph relates to the ROG informally, as follows:

- **Object soundness:** Each object ℓ in the ROG has exactly one representative OObject in the OGraph. Similarly, each domain in the ROG has exactly one representative ODomain in the OGraph. Furthermore, this mapping is consistent with respect to the ownership relation. If object ℓ is in the domain $\ell'.d$ in the ROG, then the representative of ℓ is in the representative of $\ell'.d$ in the OGraph. Similarly, if ℓ has a domain d in the ROG, then the representative for ℓ has a representative ODomain for d in the OGraph.
- **Edge soundness:** If there is a field reference from object ℓ_1 to object ℓ_2 in a ROG, then the OGraph has an OEdge between the OObjects O_1 and O_2 that are the representatives of ℓ_1 and ℓ_2, respectively.

The following Object Graph Soundness theorem restates more formally the above informal definitions, and combines *object soundness* and *edge soundness*.

Theorem: Object Graph Soundness.

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

where the $\rightsquigarrow_G^*$ relation (Fig. 3.13) is the reflexive and transitive closure of the $\rightsquigarrow_G$ relation.

By inversion of PT-APPROX, the theorem states that given a well-typed store S, an OGraph produced from the same program P, there exists a map H that maps each location ℓ in the store to a unique OObject, and a map K that maps each runtime domain in the store to a unique ODomain, and this mapping is consistent with respect to the ownership relation. In addition, the OEdges in an OGraph soundly abstract all field points-to relations between any two objects in an ROG.

To prove the Object Graph Soundness theorem, we need to show:

$$
\begin{array}{l}
\text{(1) } PtO, PtD, PtE \vdash_{CT,H} \Sigma \\
\text{(2) } (S, H, K) \sim (PtO, PtD, PtE)
\end{array}
$$

$$
\frac{}{e; S; H; K \rightsquigarrow_G^* e; S; H; K}[\text{PT-REFLEX}]
$$

$$
\frac{e; S; H; K \rightsquigarrow_G^* e''; S''; H''; K'' \qquad e''; S''; H''; K'' \rightsquigarrow_G e'; S'; H'; K'}{e; S; H; K \rightsquigarrow_G^* e'; S'; H'; K'}[\text{PT-TRANS}]
$$

Figure 3.13: Reflexive, transitive closure of the instrumented evaluation relation.

Proof. By induction on the $\rightsquigarrow^*_G$ relation. There are two cases to consider:

Case PT-REFLEX:

$(S; H; K) \sim G$	Immediate, because $S = \emptyset$
$PtO, PtD, PtE \vdash_{CT,H} \Sigma$	Immediate, from PT-SIGMA store constraint

Case PT-TRANS:

$e; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e; S; H; K$	By assumption
$(\emptyset, \emptyset, \emptyset) \sim G$	Because $S = \emptyset$
$e; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e'; S'; H'; K'$	By inversion
$(S'; H'; K') \sim G$	By induction hypothesis
$e'; S'; H'; K' \rightsquigarrow_G e; S; H; K$	By inversion
$(S; H; K) \sim G$	By Preservation

$e; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e; S; H; K$	By assumption
$(\emptyset, \emptyset, \emptyset) \sim G$	Because $S = \emptyset$
$e; \emptyset; \emptyset; \emptyset \rightsquigarrow^*_G e'; S'; H'; K'$	By inversion
$PtO, PtD, PtE \vdash_{CT,H'} \Sigma'$	By induction hypothesis
$e'; S'; H'; K' \rightsquigarrow_G e; S; H; K$	By inversion
$PtO, PtD, PtE \vdash_{CT,H} \Sigma$	By Preservation

□

3.3.7 Limitations

The proof assumes that objects are created only in locally declared domains or domain parameters. Also, it does not reflect the existence of the annotations `lent` or `unique` (Section 2.5.1, Page 65).

3.4 Display Graph (DGraph)

The static analysis extracts a hierarchical object graph, the Ownership Object Graph (OOG), from a program with ownership domain annotations. The OOG has two parts:

- OGraph: this is graph that can have cycles in the presence of recursive types;
- DGraph: this is a depth-limited unfolding of the OGraph with *lifted edges* to account for information below the cutoff depth.

3.4.1 Depth-Limited Unfolding

We do not formalize the generation of a DGraph from an OGraph. An ODomain, OObject or OEdge in an OGraph creates a corresponding DDomain, DObject or DEdge in the DGraph (Fig. 3.14). Furthermore, a DObject can merge one or more OObjects.

$$
\begin{array}{lll}
DG \in \mathsf{DGraph} & ::= \langle \textbf{Objects} = DOS,\ \textbf{Domains} = DDS,\ \textbf{Edges} = DES\ \rangle & \\
& ::= \langle DOS, DDS, DES \rangle & \\
DD \in \mathsf{DDomain} & ::= \langle\ \textbf{Id} = DD_{id}, \textbf{Domain} = d\ \rangle & \\
& ::= \langle\ DD_{id}, d\ \rangle & \\
DO \in \mathsf{DObject} & ::= \langle\ \textbf{Id} = DO_{id},\ \textbf{Types} = \{C\!<\!\overline{D}\!>\ldots\}\ \rangle & \\
& ::= \langle\ DO_{id}, \{C\!<\!\overline{D}\!>\ldots\}\ \rangle & \\
DE \in \mathsf{DEdge} & ::= \langle\ \textbf{From} = DO_{src}, \textbf{Field} = f,\ \textbf{To} = DO_{dst}\ \rangle & \\
& ::= \langle\ DO_{src}, f,\ DO_{dst}\ \rangle & \\
DOS & ::= \{DO\ldots\} & \text{Set of DObjects} \\
DDS & ::= \{DD\ldots\} & \text{Set of DDomains} \\
DES & ::= \{DE\ldots\} & \text{Set of DEdges}
\end{array}
$$

Figure 3.14: Data type declarations for the DGraph.

3.4.2 Abstraction by Types

In addition to providing abstraction by ownership hierarchy, an OOG can provide abstraction by types, as we discussed informally in Section 2.4.3.2 (Page 59). We formalize abstraction by types as a post-pass on the DGraph (Fig. 3.15). Abstraction by types relies on a heuristic based on a more flexible notion of type compatibility (Rule R-AUX-COMPAT), instead of the strict FDJ subtyping rules used in the OGraph. With the heuristic turned on, a DObject can merge OObjects that are in the same owning ODomain (Rule R-MERGE-OBJECTS).

When accounting for inheritance, domain parameters must obey the following condition:

$$C' <: C \text{ and } C'\!<\!\overline{D'}\!> <: C\!<\!\overline{D}\!> \text{ implies } \overline{D'} = \overline{D}, \overline{D''}$$

We formalize below the two heuristics, abstraction by trivial types and abstraction by design intent types.

3.4.2.1 Abstraction by trivial types

Abstraction by trivial types merges objects whenever their types share one or more non-trivial *least upper bound (LUB) types*. The heuristic does not merge objects that share only *trivial* types as supertypes. The *set* of trivial types, TT, is user-configurable, and typically includes `Object`, `Cloneable` and `Serializable` from the Java Standard Library. Many marker interfaces that do not declare any methods, such as `RandomAccess`, are also in the list.

Abstraction by trivial types corresponds to the disjunct existsNonTrivialLUB (Fig. 3.16) in R-AUX-COMPAT and can be turned-off by setting the flag byTT to false (Fig. 3.15).

SCHOLIA assumes that the program's whole source code, including external libraries that are in use, are available. Thus, the class table CT includes entries for all of those types.

[3]We formalize abstraction by types in the DGraph in order to simplify the formalization of the OGraph. We conjecture but do not prove, however, that soundness still holds when using abstraction by types.

$$\frac{\mathsf{byTT}, \mathsf{byDIT}, TT, DIT \vdash \mathsf{compat}(C,\ C') \qquad (D_1 = D_1')}{DOS, \mathsf{DObject}\langle\ DO_{id}, \{C{<}\overline{D}{>}\ldots\}\ \rangle,\ \mathsf{DObject}\langle\ DO_{id'}, \{C'{<}\overline{D'}{>}\ldots\}\ \rangle \implies DOS, \mathsf{DObject}\langle\ DO_{id''}, \{C{<}\overline{D}{>}\ldots\} \cup \{C'{<}\overline{D'}{>}\ldots\}\ \rangle}\ [\text{R-MERGE-OBJECTS}]$$

$$\frac{\begin{array}{c} C_1 <: C_2 \quad \text{or} \quad C_2 <: C_1 \\ \text{or (} \quad \mathsf{byTT} \quad \text{and} \quad TT \vdash \mathsf{existsNonTrivialLUB}(C_1,\ C_2) \quad \text{)} \\ \text{or (} \quad \mathsf{byDIT} \quad \text{and} \quad DIT \vdash \mathsf{mapToSameDIT}(C_1,\ C_2) \quad \text{)} \end{array}}{\mathsf{byTT}, \mathsf{byDIT}, TT, DIT \vdash \mathsf{compat}(C_1,\ C_2)}\ [\text{R-AUX-COMPAT}]$$

Figure 3.15: Rules for abstraction by types.

$$\frac{\exists C \in CT.(C_1 <: C \qquad C_2 <: C \qquad C \notin TT)}{TT \vdash \mathsf{existsNonTrivialLUB}(C_1,\ C_2)}\ [\text{R-ABSTRACTBY-TT}]$$

Figure 3.16: Abstraction by trivial types.

$$\frac{\exists C \in DIT.(\ C_1 <: C \qquad C_2 <: C)}{DIT \vdash \mathsf{mapToSameDIT}(C_1,\ C_2)}\ [\text{R-ABSTRACTBY-DIT}]$$

Figure 3.17: Abstraction by design intent types.

3.4.2.2 Abstraction by design intent types

Abstraction by design intent types corresponds to the disjunct mapToSameDIT (Fig. 3.17) in R-AUX-COMPAT and can be turned-off by setting the flag byDIT to false (Fig. 3.15).

In this heuristic, the developer defines an ordered list of design intent types (DIT). To decide whether to merge two objects of type C_1 and C_2, the analysis finds the first type in the DIT, $\hat{C}$, such that $C_1 <: C$ and $C_2 <: C$. If DIT does not include such a type, then this heuristic does not apply.

3.4.2.3 Abstraction by types and soundness

Abstraction by types leads only to additional merging of objects in a domain, so it does not compromise soundness. Thus, we need not prove soundness of the DGraph.

3.5 Implementation

This section discusses some implementation details.

3.5.1 Traceability

In our implementation of the OGraph, an ODomain knows about the underlying domain declaration in the code, and similarly, an OObject knows about the underlying field declarations in the code. In addition, the implementation sets the traceability information in the DGraph based on the information in the OGraph. This allows a developer using the tools to trace from the DGraph to the corresponding lines of code. For example, a developer can trace from a DObject to the corresponding new expressions in the code, and similarly, from a DEdge to the corresponding field declaration.

3.5.2 Differences between the formal and the concrete systems

There are several differences between the formal system and the concrete implementation. The formal system lacks the following language features:

- Generic types—they are implicitly supported, rather than explicitly formalized as in Generic Universe Types (Dietl et al. 2007);
- Method domain parameters;
- Arrays;
- Interfaces;
- Domain paths;
- Inner classes;
- `lent` and `unique` annotations.

The concrete system handles all of the above language features.

3.6 Discussion

3.6.1 Our Previous Formalizations

To my knowledge, this is the first time that a whole-program analysis is formalized using a constraint-based specification, with Featherweight Java. Usually, constraint-based specifications are inter- or intra-procedural, and deal with three-address code representations.

I now discuss the differences between the formalization of the static analysis in this chapter and our previous ones (Abi-Antoun and Aldrich 2007b, 2009a).

3.6.1.1 Pseudo-code

(Abi-Antoun and Aldrich 2007b) presented an early version of the object graph extraction analysis using pseudo-code, which made proving soundness unclear. In addition, in that version of the algorithm, multiple interface inheritance could potentially trigger unsoundness. To address this unsoundness, the later version added the Rule R-MERGE-EXISTING to merge objects after the fact (Abi-Antoun and Aldrich 2009a). Furthermore, when pulling objects from formal domain parameters to actual domains, the earlier algorithm added more edges than soundness required and was thus less precise.

3.6.1.2 Term-rewriting system

(Abi-Antoun and Aldrich 2009a) formalized an earlier extraction static analysis using rewriting rules. The earlier formalization proved *unique object and domain representatives* on an intermediate cyclic representation, which is then projected into a graph that is displayed. However, it was unclear that the unfolding step preserved the soundness invariants. Moreover, the earlier formalization lacked a proof of *edge soundness*.

The rewriting rules created a single AbstractDomain for each domain declaration d in the abstract syntax. In this formalization, the OGraph can already distinguish between two ODomains that have the same underlying domain d in the abstract syntax.

In the present formalization, the analysis still unfolds a cyclic OGraph to a certain threshold. The developer sees the DGraph above the threshold, and the OGraph below the threshold is still cyclic. This side-steps the issue of determining a depth at which to cutoff the recursion and the potential unsoundness of selecting an incorrect depth in the earlier representation. (Abi-Antoun and Aldrich 2009a) only conjectured and did not prove the existence of such a depth.

Also, (Abi-Antoun and Aldrich 2009a) conjectured edge soundness. Using the constraint-based specification in this chapter, we proved both *object soundness* and *edge soundness* (Section 3.3.6), which subsume the *unique object and domain representatives* and *edge soundness* defined in (Abi-Antoun and Aldrich 2009a).

Finally, using abstract interpretation makes the analysis more comparable to previous Andersen-style points-to analyses (Pichardie 2008).

3.6.2 Precision

For simplicity, the formal system does not model field updates. Indeed, initializing a field has the same challenges as assignment in our system, and the rules are no different. Still, modeling field updates and `null` could increase the precision. For instance, if a field f is never assigned to and remains null in any program run, the analysis may not create an edge. In the current model, if a class C declares a field f of type T, then the constructor must initialize the field, and the analysis conservatively assumes that an object c of type C has a points-to edge to an object t of type T.

3.6.3 Points-to Analysis

The object graph extraction analysis is a kind of a points-to analysis — a fundamental static analysis to determine the set of objects whose addresses may be stored in variables or fields of objects. A common idea in points-to analysis is to merge all the objects that are created at the same allocation site into an equivalence class. A basic points-to analysis attaches an allocation label $h \in H$ at each instruction `new` $C()$, as in:

$$\texttt{new}^h\ C()$$

The static object name is then defined as $O = H$ (Pichardie 2008). In contrast, our analysis distinguishes between allocations in different domains and that have different domain parameters,

and must analyze expressions of the kind:

$$\texttt{new}\ C{<}P_{owner}, P_{params}...{>}()$$

where P_{owner} is the owning domain, and P_{params} are optional additional domain parameters. Each of the P_i could be a formal domain parameter. At runtime, each domain parameter is bound to some actual domain, so the static analysis must track the bindings of formal parameters to actual domains.

Our static analysis is similar to a flow-insensitive Andersen-style points-to analysis (Andersen 1994), but adapted to object-oriented code (Milanova et al. 2005). The state-of-the-art is *object-sensitive* analysis (Milanova et al. 2005), particularly when computing a complete points-to solution for all the variables in a program. In contrast, a refinement-based approach, which performs points-to analysis on demand (Sridharan et al. 2005; Sridharan and Bodík 2006; Xu and Rountev 2008), may achieve higher precision, but may not scale when computing solutions for a large number of variables. Thus, a refinement-based analysis does not seem suitable for SCHOLIA which computes points-to information for an entire program.

Our analysis is object-insensitive but can considered *domain-sensitive*, since it distinguishes between objects in different domains. Since domains are coarser-grained than objects, we believe our analysis is more scalable than an object-sensitive one. However, our analysis suffers from some of the imprecisions that object-sensitivity addresses such as field assignment through a superclass (Milanova et al. 2005) (see examples of imprecision in Section 2.6.3, Page 69).

3.7 Summary

In this chapter, I formalized a static analysis to extract from a program with ownership domain annotations, a *global* hierarchical object graph. The object graph conveys architectural abstraction by ownership hierarchy and by types. Moreover, I proved that the extracted object graph is both *object sound* and *edge sound*. These properties are crucial to ensure that an extracted object graph shows all runtime objects and relations, in order to use it to analyze communication integrity.

Credits

Lecture notes by David Pichardie on the soundness of an Andersen-style points-to analysis (Pichardie 2008) inspired our style of proving soundness. Pichardie, however, used an object-oriented $While_O$ language with three-address code, rather than Featherweight Java.

Chapter 4

Evaluation of the Object Graph Extraction[1]

In this chapter, I evaluate the annotations and the static analysis by extracting hierarchical object graphs from several real representative object-oriented systems that I annotated manually.

4.1 Introduction

This chapter focuses on extracting hierarchical object graphs, and does not represent the output as a standard runtime architecture. As I mentioned in Chapter 1, however, abstracting an object graph into a C&C view is largely automatic. So we will use the terms "runtime architecture", "component" and "tier", interchangeably with "object graph", "object" and "domain", respectively.

This chapter is organized as follows. In Section 4.2, I list the research questions that this evaluation aims to answer. In Section 4.3, I discuss the tool support for the annotations and the object graph extraction. In Section 4.4, I discuss the extraction methodology. In Section 4.5, I discuss the evaluation methodology. Section 4.6 discusses a case study using the JHotDraw system. Section 4.7 discusses a case study using the HillClimber system. Section 4.8 discusses a field study using the LbGrid system. Section 4.9 has an evaluation based on a cognitive dimensions framework. I conclude this chapter with a discussion in Section 4.10.

4.2 Research Questions

Our evaluation aims to answer the following hypotheses (Section 1.10, Page 25):

> *H-1: Lightweight typecheckable ownership annotations can specify, within the code, local hints about object encapsulation, logical containment and architectural tiers.*
>
> *H-2: In practice, a static analysis can extract from an annotated program a global, hierarchical object graph that provides architectural abstraction by ownership hierarchy and by types.*

[1] Portions of this chapter appeared in (Abi-Antoun and Aldrich 2007a, 2008b, 2009a).

We refine the hypotheses into the following research questions:

RQ1 – Precision: In practice, does the static analysis, by abstracting objects to domains and types, produce object graphs that have sufficient precision? Or does it produce object graphs that suffer from being over-conservative approximations that are fully connected graphs, or collapse all the objects in a domain to a single object, in the absence of aliasing information more precise than what ownership annotations provide?

RQ2 – Abstraction by ownership: In practice, can a hierarchical object graph show architecturally relevant objects from the application domain in the top-level domains, and low-level objects that are data structures underneath architecturally significant objects?

RQ3 – Abstraction by types: In practice, can abstraction by types achieve additional architectural abstraction in an object graph?

RQ4 – Iteration: In practice, can one effectively iterate the process of adding the ownership annotations and setting the optional input to the static analysis, e.g., to control abstraction by types, to extract an object graph with the desired architectural abstraction?

RQ5 – Annotations: Do the annotations describe local, modular information regarding object encapsulation, logical containment and architectural tiers? Or does a developer adding the annotations need some high-level global information?

RQ6 – Value: In practice, does an OOG highlight potentially useful information about the system's runtime structure?

4.3 Tool Support

The tool support for extracting object graphs consists of two plugins in the Eclipse open source development environment, which has become popular with researchers and practitioners (Goth 2005; Murphy et al. 2006). The first tool manages and typechecks the annotations and the other one extracts and displays an object graph from an annotated program.

4.3.1 Annotation Tool

I designed a set of Java 1.5 annotations that implement the ownership domain type system using existing language support for annotations. I also re-implemented a typechecker for the annotations, ArchCheckJ, which stands for Architectural annotation Checker for Java. ArchCheckJ is a plugin to the Eclipse Java Development perspective (JDT), and displays annotation warnings in the Eclipse problem window. A developer can double-click on a warning in the problem window to go the line of code with the missing or inconsistent annotation. Additional details on the annotation language and the design of ArchCheckJ are in Appendix A.

4.3.2 Object Graph Extraction Tool

I implemented the static analysis to extract an object graph (which we discussed in Chapters 2, 3) as another Eclipse plugin, ArchRecJ, which stands for Architectural Recovery for Java. The object graph extraction works in the presence of annotation warnings, but warns that the extracted object graph may not soundly reflect all objects and relations.

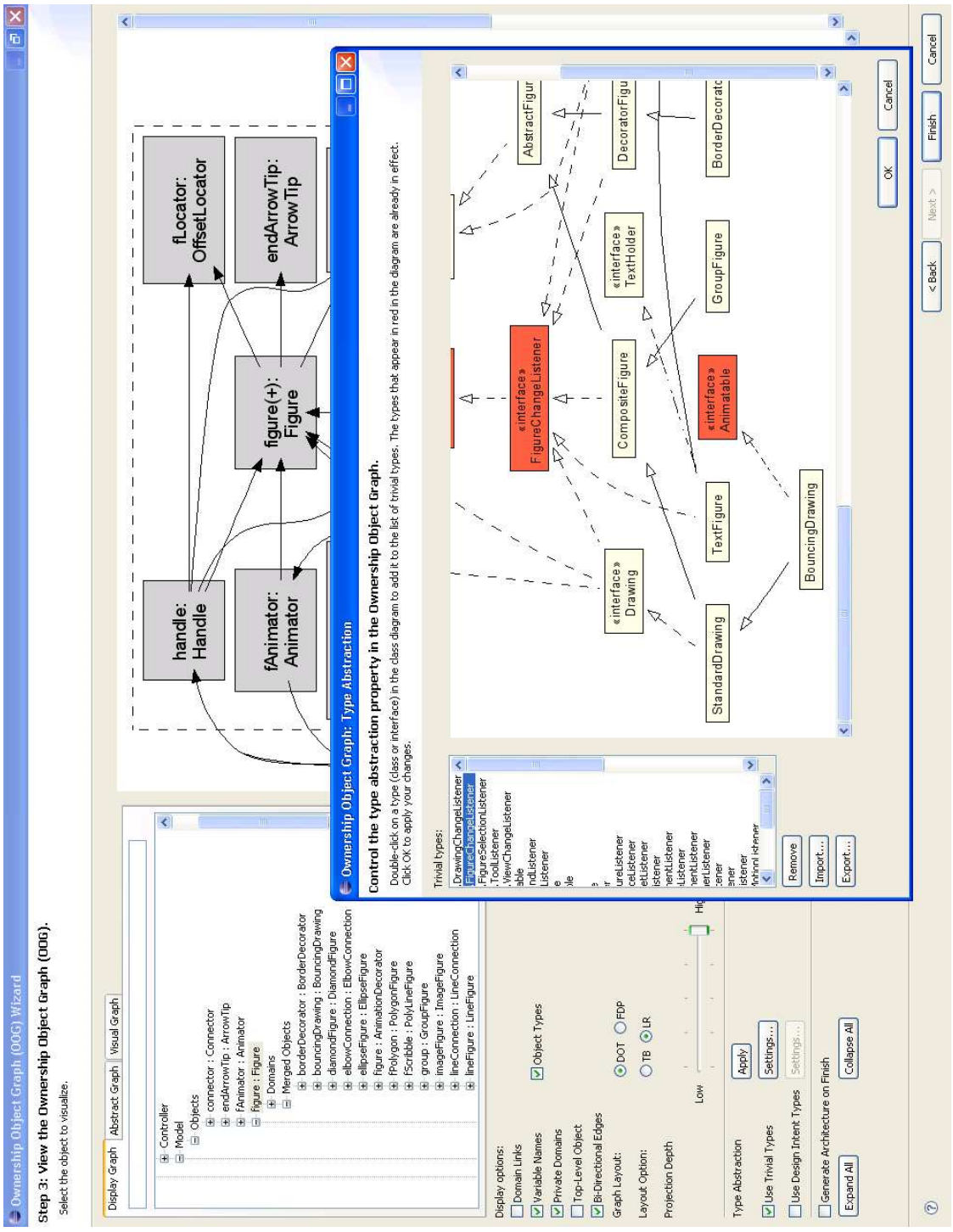


Figure 4.1: ArchRecJ tool: the left pane shows the ownership tree and the right pane shows the depth-limited unfolding. The tool shows the field declarations that an object in the OOG represents. The tool also helps a developer select the trivial types and the design intent types for the abstraction by types.

The ArchRecJ tool offers the following features (Fig. 4.1):

- **Select top-level object:** the user can interactively select an object as the root of the graph to view its substructure;
- **Set trivial types:** a developer can specify an optional list of trivial types to use the abstraction by types feature;
- **Set design intent types:** a developer can specify an optional list of design intent types to use the abstraction by types feature;
- **Display inheritance hierarchy:** the tool can display the inheritance hierarchy of the types of the field declarations that a display object merges, to help the developer fine-tune the list of trivial types or design intent types for the abstraction by types;
- **Collapse or expand selected item:** a developer can collapse or expand the sub-structure of a selected object or domain;
- **Control unfolding depth:** a developer can control the visible depth of the ownership tree, using the slider control in Fig. 4.1;
- **Set object labels:** Each object in an extracted object graph represents at least one field or variable declaration in the program. An object might have multiple types, and the analysis picks one of those types as the label. ArchRecJ can label objects with an optional field name or variable name and an optional type name. The type used in the label consists of a least-upper-bound type or a design intent type or a labeling type (discussed below);
- **Set additional labeling types:** the object graph extraction non-deterministically selects a label for a given object o based on the name or the type of one of the references in the program that points to o. A developer can specify an optional list of labeling types for labelling objects. For example, in Fig. 2.3(b), the tool adds the decoration `(Listener)` to an object's label, if it merges at least one object of that type, as is the case for `pieChart`, `barChart` and `model`. We implemented this feature in response to the developer's feedback during the field study, because he informed us that labels are very important in a diagram;
- **Trace to code:** the tool can show the list of field declarations and their types that a given display object merges. In addition, the developer can trace from the field declarations to the right lines of code. This feature is useful to guide the developer to the field declarations in the program that require different annotations.
- **Navigate:** the tool supports zooming in and out, panning, scrolling and other standard operations;
- **Search:** the tool supports searching for an object in the ownership tree by type or field name;
- **Persist extracted OOG:** the tool can persist an extracted OOG into an XML file. This file can then be viewed using a standalone viewer. When using the viewer, the developer cannot control the abstraction by types, but can still expand or collapse selected elements.

Thus far, our research has focused on the underlying static analysis rather than on novel techniques for visualizing object graphs. For instance, our visualization uses the simple but effective GraphViz tool (Gansner and North 2000) which supports clustered graphs, but does not support visual features such as cross-hatching fill patterns. Future work may consider using more specialized visualization frameworks such as SHRIMP VIEWS (Storey et al. 1998).

4.4 Extraction Methodology

In this section, I discuss the SCHOLIA methodology to extract object graphs. Following the general SCHOLIA approach (Section 1.7, Page 20), this involves adding and checking the annotations (Section 4.4.1), then running the static analysis (Section 4.4.2).

The study's experimenter (hereafter "I") developed the ArchCheckJ and ArchRecJ tools, but none of the subject systems. I mostly learned the architectural structure of the subject systems from iteratively annotating the code, examining the extracted OOGs and relating the OOGs to class diagrams drawn by others, or to other available documentation.

4.4.1 Adding and Checking the Annotations

In this section, I discuss the process of adding the annotations, typechecking them, and addressing the annotation warnings.

4.4.1.1 Gathering available documentation.

Before adding annotations and extracting object graphs, it is often useful to have an informal diagram of the target architecture, to help guide the annotation process. Indeed, most architectural extraction case studies start by gathering available documentation (Tzerpos and Holt 1996). When available, the documentation can help identify the domains in the system, the types that are most architecturally relevant and the hierarchical system decomposition, i.e., how to decompose some of the objects into nested sub-structures.

For example, for the JHotDraw system, I had access to a tutorial by JHotDraw's original designers (Beck and Gamma 1997; Gamma 1998), but for a slightly older version than the version of JHotDraw I annotated. One of the tutorials discusses the design patterns that JHotDraw implements, using a code architecture, but does not describe JHotDraw's runtime structure. I also found several class diagrams drawn by others who studied JHotDraw, e.g., (Riehle 2000).

4.4.1.2 Typechecking the annotations

A developer adding the annotations often follows an iterative process. After each round of annotations, he runs the typechecker, examines the warnings, and addresses them from the most to the least important ones.

The annotations are modular and can be checked one class at a time. However, some amount of iteration is involved. For instance, if the developer defines a domain parameter on a class, she has to find all the locations in the code that use that class, and bind that domain parameter to some other domain in scope. So this may require a continuous annotate-check cycle.

4.4.1.3 Prioritizing the annotation warnings

It is often helpful to fix the annotation warnings in a specific order. I illustrate these using the Listeners example (Fig. 2.4). From most to least important are:

1. Undeclared domains or domain parameters. For instance, the domain `OWNED` must be declared (line 5 in Fig. 2.4) before the `listeners` field declaration can be annotated with `OWNED` (line 9). Similarly, the domain parameters `M` (line 21) and `V` (line 3) must be declared.
2. Unbound domain parameters at field and variable declarations. For instance, since class `Model` takes a domain parameter `V`, the field declaration `model` of type `Model` must bind the domain parameter to another domain in scope, e.g., `VIEW` (line 29). This also includes binding the domain parameters on containers such as `Vector` and `List` (lines 9, 24). Recall that `List` takes an `ELTS` formal domain parameter for the list elements.
3. Domain parameter inheritance. For instance, the domain parameter `M` on `PieChart` is bound to the domain parameter on `PieChart`'s superclass, `BaseChart` (line 19).
4. Assignment rule. For instance, a reference annotated with `DOCUMENT` cannot be assigned to another one annotated with `VIEW`. Similarly, a `lent` variable, which denotes a temporary alias, cannot be stored in a field.
5. Array parameters. Domain annotations for the array elements must be also provided.
6. External annotation files. The ArchCheckJ tool allows a developer to partially annotate the parts of the Java standard library or other third-party libraries that are in use (Section A.4.1, Page 324). The external files for the Java Standard Library can often be reused across different systems.
7. Domain links. Finally, a developer can set domain links and link assumptions to enforce access permissions between domains.

4.4.2 Refining the Object Graph

In this section, I discuss the process of refining the extracted object graph using annotations.

4.4.2.1 Overall strategy

Just as there are multiple architectural views of a system, there is no single right way to annotate a program. And different annotations can produce different object graphs (Refer to discussion in Section 2.6.2, Page 66). However, a type system ensures that the annotations are consistent with each other and with the code.

Good annotations minimize the number of objects in the top-level domains by pushing more objects underneath other objects. In particular, the goal is to remove from the top-level domains low-level objects that are data structures, such as instances of `Vector` and `List`. Ideally, the top-level domains show only objects that are architecturally relevant and correspond to entities from the application domain.

4.4.2.2 Refining the ownership annotations

A developer controls the architectural extraction process as follows. First, she chooses the top-level domains. Then, she achieves the desired number of objects in each top-level domain, primarily through *abstraction by ownership hierarchy*.

A developer-specified annotation can push an object underneath—i.e., into a private or a public domain declared inside—a more architecturally-relevant object. The parent object becomes *primary*, and the child object becomes *secondary*. As a result, only primary objects appear in the top-level domains. Each of those objects has more domains and objects, until low-level objects are reached. In addition, the developer must minimize the remaining annotation warnings, especially any high-priority ones.

To summarize, a developer can do any of the following: (a) Push a secondary object underneath a primary object using the strict encapsulation of private domains; (b) Push a secondary object underneath a primary object using the logical containment of public domains; or (c) Pass a low-level object linearly using the `unique` annotation.

4.4.2.3 Code changes

In some cases, adding annotations that specify strict encapsulation and avoid the representation exposure may require a change to the code, e.g., to return a copy of an internal list instead of an alias (Aldrich et al. 2002c; Aldrich and Chambers 2004). Generally, using logical containment does not require any code changes. In most cases, defining public domains required changing the annotations only locally and incrementally.

4.4.2.4 Using abstraction by types

To reduce clutter further, the developer can enable *abstraction by types*, which merges more objects in a given domain, based on the architectural relevance of their declared types. The object graph extraction tool provides some support to help a developer select the types to be used for abstraction by types.

4.4.2.5 Controlling the level of detail

Finally, she achieves an appropriate level of visual detail by expanding or collapsing the substructure of selected objects, or changing the unfolding depth uniformly across the graph. The analysis adds any lifted edges to account for the elided substructures.

4.5 Evaluation Methodology

The evaluation methodology follows closely SCHOLIA's extraction methodology above (Section 4.4), and involves the following steps:

1. Add annotations to the code and typecheck them;
2. Extract an object graph that conveys architectural abstraction by ownership hierarchy. Optionally, specify types to control abstraction by types;
3. Iterate the annotations, and the abstraction by types.

In addition, in preparation for the evaluation, I performed the following tasks, which may not be always required.

Making minor code changes to use annotations. In most cases, we did not change the code as we were adding the annotations. However, we made some minor code changes as required by the annotation system[2]. For instance, one change may involve extracting a local variable from a new expression, in order to add an annotation on the local variable. Another change is to convert an anonymous class to a nested class, in order to declare domain parameters on the class.

Refactoring to generics. Two subject systems, JHotDraw (Section 4.6) and HillClimber (Section 4.7), were developed prior to Java 1.5 and did not use generic types. I refactored them to use generics, mostly automatically using Eclipse's tool support (Fuhrer et al. 2005). The LbGrid subject system in the field study (Section 4.8) was already using Java 1.5 and generic types and did not require such a refactoring.

Re-engineering system during annotation process. I had previously studied the HillClimber subject system when I re-engineered it to ArchJava (Abi-Antoun et al. 2007a). The re-engineering study also produced a version that cleaned up the original code, for instance by making most class fields be private. For this case study, I started from the refactored Java version and added ownership domain annotations to it.

4.6 Extended Example: JHotDraw

JHotDraw (JHD 1996) is open source, rich with design patterns, uses composition and inheritance heavily and has evolved through several versions. For this case study, we used version 5.3, which has around 200 classes and 15,000 lines of Java.

Design documentation for JHotDraw is available, e.g,. (Gamma 1998; Riehle 2000; Kaiser 2001). A manually drawn class diagram (Fig. 4.2) shows some of the core types. An often-cited article (Kaiser 2001) discusses that JHotDraw follows the Model-View-Controller design pattern. However, the JHotDraw package structure does not reveal that fact, since all the core types are in one `framework` package.

4.6.1 Annotation Process

In this section, I discuss the process of adding annotations to JHotDraw to explain what the annotations look like and the information that they describe. In particular, a developer focuses mainly on the structure of the system, rather than its behavior. Moreover, when adding the annotations, the developer describes only local, modular information, and does not require direct knowledge of the global system structure.

[2]One proposal, JSR 308 (Ernst and Coward 2006), permits annotations to appear in more places, such as on generic type arguments. Some of the code changes we made may no longer be necessary once JSR 308 is adopted into the Java language, and supported by existing development environments such as Eclipse.

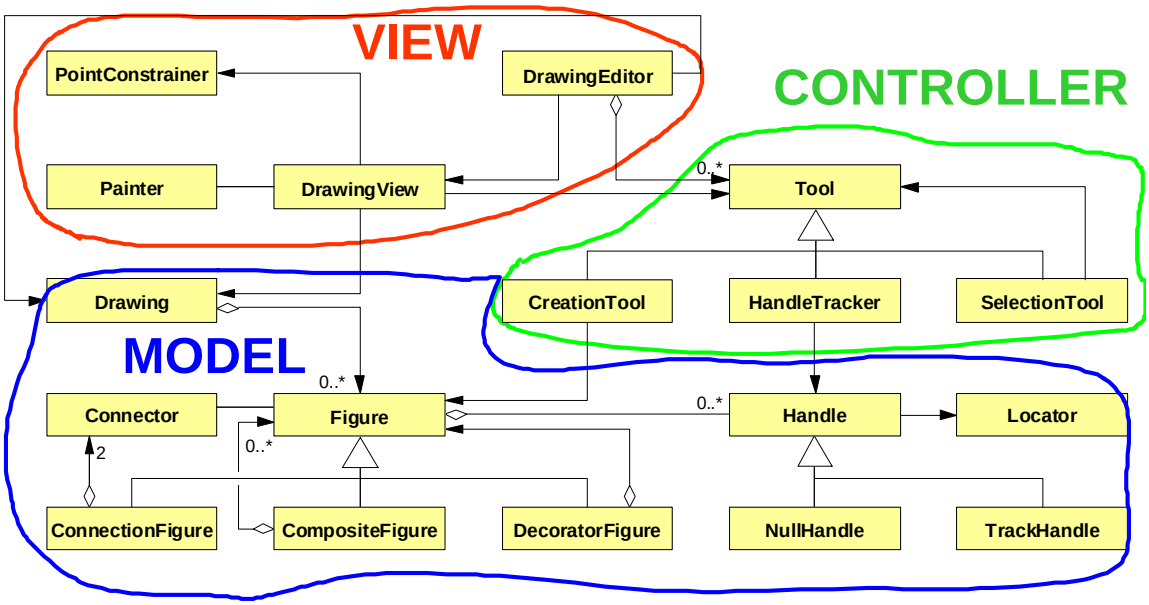


Figure 4.2: JHotDraw class diagram showing how we annotated instances of the selected types. Source: (Riehle 2000).

4.6.1.1 Annotation Overview

For JHotDraw, I defined the following three top-level domains and organized instances of the core types as follows:

- `MODEL`: has instances of `Drawing`, `Figure`, `Handle`, etc. A `Drawing` is composed of `Figures` that know their containing `Drawing`. A `Figure` object has `Handles` for user interactions. The `Drawing` interface also extends `FigureChangeListener` (not shown in Fig. 4.2) to listen to changes to its `Figures`.
- `VIEW`: has instances of `DrawingEditor`, `DrawingView`, etc. The `DrawingView` class extends `DrawingChangeListener` (not shown) to listen to changes to `Drawing` objects;
- `CONTROLLER`: has instances of `Tool`, `Command` and `Undoable`. A `DrawingView` uses a `Tool` to manipulate a `Drawing`. A `Command` encapsulates an action to be executed, i.e., implements the Command design pattern without undo.

Once I defined the three top-level ownership domains, `MODEL`, `VIEW` and `CONTROLLER`, I parameterized most of the JHotDraw types with the corresponding domain parameters, `M`, `V` and `C`, respectively. Some of these types required only one or two of `M`, `V` and `C`. I could have further reduced these parameters by using the implicit owner domain parameter, accessible using the `OWNER` annotation[3].

4.6.1.2 Annotation Examples and Observations

In the following discussion, I illustrate the annotation process using actual examples and code snippets from JHotDraw. I slightly edited the code for presentation by removing the trivial

[3]We did not use the `OWNER` annotation initially (Section 2.3.3, Page 41), because the tools did not fully support it at the time. In future work, we will update the annotated subject systems to use the `OWNER` annotation more heavily.

```
class DrawApplication<M,V,C> ... implements DrawingEditor<M,V,C> ... {
}

class MDI_DrawApplication<M,V,C> extends DrawApplication<M,V,C> ... {
}

class JavaDrawApp<M,V,C> extends MDI_DrawApplication<M,V,C> {
}

class Main {
  domain MODEL, VIEW, CONTROLLER;
  ...
  VIEW JavaDrawApp<MODEL,VIEW,CONTROLLER> app = new JavaDrawApp();

  public void run() {
    app.open();
  }

  public static void main(lent String args[shared]) {
    lent Main system = new Main();
    system.run();
  }
}
```

Figure 4.3: JHotDraw: defining the three top-level domains on the root class.

visibility modifiers such as `private` or `public` [4]. I make several observations based on studying the annotations.

Observation: Ownership domains specify architectural runtime tiers. A tiered architecture is often used to organize an application into a User Interface tier, a Business Logic tier, and a Data tier. Ownership domains express a tiered runtime architecture by representing a tier as an ownership domain (Aldrich and Chambers 2004), and a permission between tiers as a domain link to allow objects in the User Interface tier to refer to objects in the Business Logic tier but not vice versa. Such an architectural structure and constraints cannot be expressed in plain Java code. For example, I organized the JHotDraw runtime structure according to the Model-View-Controller design pattern (Fig. 4.3).

Observation: Ownership domains enforce instance encapsulation. All ownership type systems can express and enforce instance encapsulation which is stronger than the module visibility mechanism of making a field `private`. In ownership domains, placing a field in the private `OWNED` domain means that the object can be reached only by going through its owner. As a result, no aliases to that object can leak to the outside.

Consider `CompositeFigure` in JHotDraw (Fig. 4.4). Placing the list of composite `Figures`, represented by the field `fFigures`, in the `OWNED` private domain encapsulates `fFigures` to prevent objects that only have access to the composite object from modifying the list directly. If a developer tries to subvert the language visibility mechanisms by returning a reference to a `private`

[4](Abi-Antoun and Aldrich 2007a) shows mostly the same examples, but in the concrete annotation language.

```
/**
 * The interface of a graphical figure. A figure knows its display box
 * and can draw itself. A figure can be composed of several figures.
 * A figure has a set of handles to manipulate its shape or attributes.
 * A figure has one or more connectors that define
 * how to locate a connection point.
*/
interface Figure<M> extends Storable <M> {
  ...
}
/**
 * A Figure that is composed of several figures.
 */
abstract class CompositeFigure<M> extends AbstractFigure<M>
                                  implements FigureChangeListener<M> {

  domain OWNED;
  /**
   * The figures that this figure is composed of
   */
  OWNED Vector<M Figure<M> > fFigures;

  /**
   * Adds a vector of figures.
   */
  void addAll(M Vector<M Figure<M>> newFigures) {
    // Cannot assign object M Vector newFigures to owned Vector fFigures
    // this.fFigures = newFigures;
    fFigures.addAll(newFigures);
  }
}
```

Figure 4.4: JHotDraw: `CompositeFigure` annotations.

or `protected` field using a `public` accessor method, the typechecker prohibits a `public` method from taking an `OWNED` parameter or returning an `OWNED` object.

For example, during software evolution, a novice developer can use Eclipse to generate a setter for the `fFigures` field. Eclipse produces the following code, without annotations:

```
void setFFigures(Vector<Figure> figs) {
  this.fFigures = figs;
}
```

As the developer is adding the annotations to the `setFFigures()` method, the typechecker can warn him that the parameter `figs` of a non-private method cannot be marked as `OWNED`. And any other annotation would fail the assignment check when overwriting the `fFigures` field.

To avoid the warning, the developer can rewrite the `setFFigures()` method to no longer overwrite the existing field, and instead, call the `clear()` and `allAll()` methods.

```
void setFFigures(Vector<Figure> figs) {
  // Use the following, instead of overwriting the field
  this.fFigures.clear();
```

```
    this.fFigures.addAll(figs);
  }
```

When manually adding annotations, it is possible to miss many opportunities for strictly encapsulating objects. Indeed, I initially annotated `fFigures` with the domain parameter `M` instead of the `OWNED` domain. In many cases, objects should be encapsulated to avoid the representation exposure, but are not. Making these objects encapsulated may require a code change, e.g., by returning a shallow copy of an object such as a `List`, instead of an alias.

Extracting the object graph helped visualize the annotations and encouraged the use of strict encapsulation since `OWNED` objects no longer clutter the top-level domains. Future work may include developing a tool to prompt a developer when a field could be encapsulated. For example, a lightweight compile time ownership inference algorithm, e.g., (Liu and Milanova 2007), could suggest possible Eclipse "quickfixes" to strictly encapsulate objects.

Observation: Ownership domains expose implicit communication. Design patterns such as Observer (Gamma et al. 1994, p. 293) can decouple object-oriented code, but tend to make the communication between objects implicit. Adding ownership domain annotations can help make that communication more explicit.

We initially wanted to parameterize `Drawing` (Fig. 4.5) with only the `M` domain parameter, but `DrawingChangeListener` is implemented by `DrawingView`. So the `DrawingChangeListener` reference had to be in the `VIEW` domain, which in turn required the `V` domain parameter. By making implicit communication explicit, the annotations seem to prematurely constrain `DrawingChangeListener` objects to be in the `VIEW` domain. Since `Drawing` was a core interface referenced by other interfaces in the core `framework` package, this led to passing all three domain parameters to many additional interfaces and classes that implement those interfaces.

If `Drawing` did not have to be parameterized by domain parameter `V`, I might not have discovered the implicit communication in the observer by adding the annotations. Thus, ownership domain annotations can help make implicit communication explicit, when a reference requires permission to access a new part of the program for the first time.

Observation: Ownership domains expose tight coupling. Let us temporarily ignore the earlier limitation with adding annotations to the listeners and assume that `Drawing` could be parameterized by only the `M` domain parameter. Let us now consider whether it would be possible to parameterize interface `Handle` (Fig. 4.6) with domain parameter `M` and `C`. A `Handle` would be in the `C` domain parameter and access objects in that domain parameter and in the `M` domain parameter, i.e., it should not access objects in the `V` domain parameter. Note that even if the explicit parameter `C` was not provided, that domain would still be accessible to `Handle` using the implicit `OWNER` annotation.

A comment in the code indicated that Version 4.1 deprecated the original `invokeStart()` method which took a `Drawing` object as one of its parameters, in favor of an `invokeStart()` method that takes instead a method parameter of type `DrawingView`, which is parameterized by `M`, `V` and `C`. This required passing to `Handle` the additional domain parameter `V`. Since `Handle` is a core interface referenced by other interfaces in the core `framework` package, this also led to passing all three domain parameters to many additional types.

```
/**
 * Drawing is a container for figures. Drawing sends out DrawingChanged
 * events to DrawingChangeListeners whenever a part of its area was
 * invalidated. The Observer pattern is used to decouple the Drawing
 * from its views and to enable multiple views.
 */
interface Drawing<M,V> ...{

 /**
  * Adds a listener for this drawing.
  * DrawingView implements DrawingChangeListener,
  * so the objects are in 'V domain parameter
  */
  void addDrawingChangeListener(V DrawingChangeListener<M,V> listener);

  /**
  * Adds a figure and sets its container to refer to this drawing.
  * @param figure to be added to the drawing
  * @return the figure that was inserted (might be different from the figure specified).
  */
  M Figure<M> add(M Figure<M> figure)
}
```

Figure 4.5: JHotDraw: adding annotations to `Drawing`.

```
/**
 * Handles are used to change a figure by direct manipulation.
 * Handles know their owning figure and they provide methods to locate
 * the handle on the figure and to track changes.
 * Handles adapt the operations to manipulate a figure to a common interface.
*/
interface Handle<M,V,C> {

  /**
   * @deprecated As of version 4.1, use invokeStart(x, y, drawingView)
   */
  void invokeStart(int x, int y, lent Drawing<M> drawing);

  /**
   * Tracks the start of the interaction.
   * @param x the x position where the interaction started
   * @param y the y position where the interaction started
   * @param view the handles container
   */
  void invokeStart(int x, int y, V DrawingView<M,V,C> view);

  M Undoable<M,V,C> getUndoActivity();
}
```

Figure 4.6: JHotDraw: `Handle` with `M`, `V` and `C` domain parameters.

```
interface Undoable<M,V,C> {
  ...

  V DrawingView<M,V,C> getDrawingView();
}
```

Figure 4.7: JHotDraw: `Undoable` with `M`, `V` and `C` domain parameters.

```
interface Handle<M,C> {

  void invokeStart<V> (int x, int y, V DrawingView<M,V,C> view);

  M Undoable<M> getUndoActivity();

}
```

Figure 4.8: JHotDraw: `Handle` with only `M` and `C` domain parameters.

Observation: Ownership domains expose and enforce object borrowing. Let us assume that the above refactoring after JHotDraw Version 4.1 which introduced the tighter coupling was never performed, i.e., `Handle` still needed a `Drawing` instead of a `DrawingView`. Undo support was added to JHotDraw for the first time in Version 5.3. In particular, `Handle` now had a reference to `Undoable` —which in turn required domain parameters `M`,`V` and `C` because `Undoable`'s `getDrawingView()` method returned a `DrawingView` (Fig. 4.7).

Now, let us see if it would be possible to annotate `Undoable` and `Handle` with only the domain parameters `M` and `C` (Fig. 4.8). The domain parameter `V` can then be added to `invokeStart()` as a method domain parameter.

Using a method domain parameter to annotate the formal parameter `view` could enforce the constraint that a developer should not store in a field the `DrawingView` object that is passed as an argument to `invokeStart()` (Fig. 4.9). Of course, a developer could store the `DrawingView` object in a field of type `Object`, but that field would have to be cast to a `DrawingView` in order to be useful.

Instead of using a method domain parameter to enforce object borrowing, one could use the `lent` annotation to allow a temporary alias to an object within a method boundary. We found a few such examples in JHotDraw. For instance, the method `setAffectedFigures()` (Fig. 4.10) makes a copy of the `lent` argument because it cannot hold on to it.

In fact, the `lent` annotation can be formally modeled as a method domain parameter. The type system prohibits a method from returning a `lent` value, although it allows a method to return an object in a method domain parameter. In the case of `DrawingView`, `lent` cannot be used because implementations of `invokeStart()` construct `Undoable` objects that maintain aliases to the `DrawingView`. As a result, `Handle` requires the `V` domain parameter.

For that same reason, the `Undoable` interface requires the `V` domain parameter because `Undoable` stores the `DrawingView` in which the activity to be undone was performed, in order to undo the changes to that view only. This may slightly violate the Model-View-Controller design, where model objects should not hold on to view objects, because there might be multiple views that need to be updated in response to changes in the model. At the same time, it would be counter-intuitive for a user to undo a change in one view and observe changes in some

```
/**
 * AbstractHandle provides default implementation for Handle interface.
 */
abstract class AbstractHandle<M,C> implements Handle<M,C> {

  // The following would not typecheck since V not bound
  V DrawingView<M,V,C> view;

 /**
  * @param x the x position where the interaction started
  * @param y the y position where the interaction started
  * @param view the handles container
  */
 void invokeStart<V>(int x, int y, V DrawingView<M,V,C> view) {
   // Cannot store argument view in field this.view
   ...
 }
}
```

Figure 4.9: JHotDraw: using method domain parameters to enforce object borrowing.

other view. Thus, ownership domain annotations expose the tighter coupling that the Undo feature introduced. Fig. 4.10 shows in more detail the interaction between `Handle`, `Undoable` and `DrawingView`.

An earlier empirical study of JHotDraw mentioned that "a common architectural mistake [...] was to provide `Figures` with a reference to the `Drawing` or the `DrawingView`. `Figures` do not by default have any access to either the `Drawing` or the `DrawingView` in which they are contained. This prevents them from accessing information such as the size of the `Drawing`. However, it is possible to overcome this problem by passing the view into the constructor of a figure, which can then store and access this as required" (Kirk et al. 2006). Due to the stronger coupling in Version 5.3, one could now get to the `Figure`'s `Handles` through its `handles()` method then get a `DrawingView` through a `Handle`'s `UndoActivity` objects.

Observation: Ownership domains can help identify singletons. While adding ownership domain annotations, we discovered a curious instance of the Singleton design pattern: `IconKit`'s constructor was not private, although it had a static `instance()` method. Indeed, there is a `unique` instance of `DrawingEditor` (the application itself) and a `unique IconKit` (Fig. 4.11) at runtime.

4.6.1.3 Expressiveness Challenges

Like any type system, the ownership domain type system has some expressiveness challenges, that make it rule out presumably valid programs. In this section, I discuss some expressiveness challenges I encountered while adding the annotations. Some of these challenges had been previously mentioned in the ownership types literature, e.g. (Schäfer and Poetzsch-Heffter 2007).

```
class ResizeHandle<M,V,C> extends LocatorHandle<M,V,C> {
  @Override
  void invokeStart(int x, int y, V DrawingView<M,V,C> view) {
    setUndoActivity(createUndoActivity(view));
    ...
  }
/**
  * Factory method for undo activity. To be overriden by subclasses.
  */
  M Undoable<M,V,C> createUndoActivity(V DrawingView<M,V,C> view) {
    unique ResizeHandle.UndoActivity<M,V,C> undoActivity = new ResizeHandle.UndoActivity(view);
    return undoActivity;
  }
  static class UndoActivity<M,V,C> extends UndoableAdapter<M,V,C> {
     UndoActivity(V DrawingView<M,V,C> newView) {
      super(newView);
      ...}
  }
}

class UndoableAdapter<M,V,C> implements Undoable<M,V,C> {
  OWNED Vector<M Figure> myAffectedFigures;
  V DrawingView<M,V,C> myDrawingView;

  UndoableAdapter(V DrawingView<M,V,C> newDrawingView) {
    myDrawingView = newDrawingView;
  }
  void setAffectedFigures(lent FigureEnumeration<M> newAffectedFigures) {
    // the enumeration is not reusable therefore a copy is made
    // to be able to undo-redo the command several time
    rememberFigures(newAffectedFigures);
  }
  void rememberFigures(lent FigureEnumeration<M> toBeRemembered) {
    myAffectedFigures = new Vector<Figure>();
    myAffectedFiguresCount = 0;
    while (toBeRemembered.hasMoreElements()) {
      myAffectedFigures.addElement(toBeRemembered.nextElement());
      myAffectedFiguresCount++;
    }
  }
}
```

Figure 4.10: JHotDraw: concrete implementation class of `Handle`.

Observation: One object cannot be in more than one ownership domain. Ownership domains, as most other ownership type systems, support only *single ownership*, i.e., an object cannot be part of more than one ownership hierarchy. Proposals for *multiple ownership* (Cameron et al. 2007) lift this restriction in other type systems. Ownership domains do not support *ownership transfer* (Müller and Rudich 2007) either, i.e., an object's owner does not change —only `unique` objects can flow between any two domains.

As a result, one cannot define many fine-grained ownership domains to represent multiple

```
class Iconkit {
 static unique Iconkit fgIconkit = null;

 /**
  * Constructs an Iconkit that uses the given editor
  * to resolve image path names.
  */
 unique
 public Iconkit(unique Component component) {
  ...
  fgIconkit = this;
 }

 /**
  * Gets the single instance
  */
 public unique static Iconkit instance() {
   return fgIconkit;
  }
}
```

Figure 4.11: JHotDraw: annotating a singleton using `unique`.

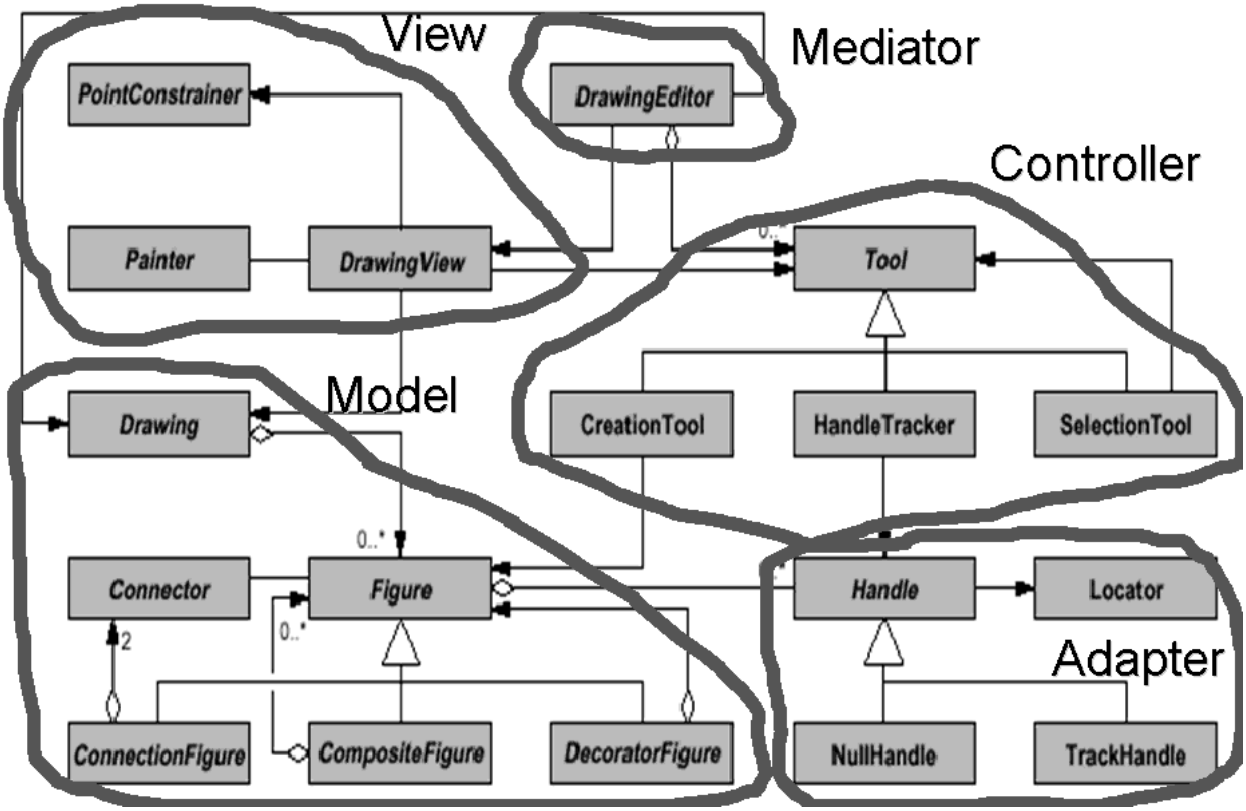


Figure 4.12: JHotDraw: alternative top-level domains. Source: (Christensen 2004).

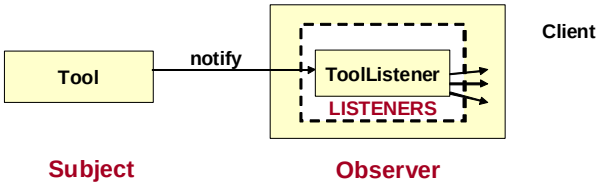


Figure 4.13: Using public domains to group objects.

roles in design patterns. For instance, (Christensen 2004) had suggested an alternative structuring of the JHotDraw types, into a Model-View-Controller-Mediator-Adapter architecture (Fig. 4.12). However, it would have been more challenging to create top-level ownership domains to correspond to such a decomposition, compared to the three top-level domains for `MODEL`, `VIEW` and `CONTROLLER` we adopted. Due to the single ownership model, placing a `DrawingEditor` object in a `MEDIATOR` domain would have prohibited it from also being in the `VIEW` domain.

Observation: An object cannot place itself in a domain it declares. An object cannot place itself in an ownership domain that it declares. This is problematic for the root application object, i.e., the `JavaDrawApp` instance (`JavaDrawApp` extends `DrawApplication` which in turn extends `DrawingEditor`). To solve this problem, we created a fake top-level class `Main` to declare the `MODEL`, `VIEW` and `CONTROLLER` top-level ownership domains, then declared the `JavaDrawApp` object in the `VIEW` domain (Fig. 4.3).

Observation: Public domains can be hard to use. Public domains make the ownership domain type system more flexible than an *owner-as-dominator* type system, e.g., (Clarke et al. 1998). Also, public domains are ideal for visualization because placing an object inside a public domain of another object relates these objects without cluttering the top-level domains. However, public domains are typically hard to use without refactoring the code. We started using them in a few cases but quickly abandoned those attempts.

Since the Observer design pattern tends to make communication between objects implicit, we attempted to represent listeners more explicitly using ownership domain annotations. For instance, it might make sense to place the `Listener` objects that an `Observer` will notify in a public domain `LISTENERS` on the `Observer`. This is because a `Listener` often needs special access to the `Observer`, but usually does not need special access to the `Subject` (Fig. 4.13).

JHotDraw uses a delegation-based event model. For instance, a `DrawingView` calls the method `figureSelectionChanged` to notify a `FigureSelectionListener` observer of any selection changes. So it might make sense to declare a `LISTENERS` public domain on `Command` to hold the `FigureSelectionListener` objects (Fig. 4.14). But the base implementation class, `AbstractCommand`, implements the `FigureSelectionListener` interface, so a `Command` *is-a* `FigureSelectionListener`. Thus a `Command` object cannot split a part of itself and place it in the public domain `LISTENERS` that it declares.

Observation: Adding annotations to listener objects can be challenging. There were additional complications when trying to highlight the event subsystem in JHotDraw using ownership

```
1 abstract class AbstractCommand<M,V,C> implements Command<M,V,C>,
2                                                   FigureSelectionListener<M,V,C> ... {
3
4     public domain LISTENERS;
5     ...
6 }
```

Figure 4.14: JHotDraw: attempting to define a public domain.

domain annotations. For example, `Command`, which is in the `CONTROLLER` domain, implements `FigureSelectionListener`, and so does `DrawingEditor`, which is in the `VIEW` domain.

Consider the method `addFigureSelectionListener()` (Fig. 4.15). How would one annotate the formal parameter `fsl` of type `FigureSelectionListener`? The parameter should support both annotations `C<M,V,C>` and `V<M,V,C>`. Indeed, the code calls `addFigureSelectionListener()`, once with a `Command` object, and another time with a `DrawingEditor` object. Currently, using either annotation for the `fsl` parameter generates an annotation warning, because one or the other method invocation would not typecheck.

Indeed, (Schäfer and Poetzsch-Heffter 2007) previously identified the difficulty of adding ownership domain annotations to programs involving listener objects and proposed a solution using a variant of the ownership domain type system. Similarly, existential ownership (Clarke 2001; Krishnaswami and Aldrich 2005; Lu and Potter 2006) could increase the expressiveness in this case. For example, (Lu and Potter 2006) would annotating the `fsl` parameter with "`any`", to typecheck both calls to `addFigureSelectionListener()`. Future work may include addressing some of these expressiveness limitations in the type system.

Observation: Adding annotations to static code can be challenging. Even a well-designed program as JHotDraw had static code, which is challenging for many ownership type systems. In particular, the static `Hashtable` cannot have the `M`, `V`, and `C` domain parameters because the domain parameters declared on the class `NullDrawingView` are not in scope for static members (Fig. 4.16). Static members can only be annotated with `shared` or `unique`, and these values cannot flow to the `Mx`, `Vx` or `Cx` method domain parameters. Currently, this code cannot be successfully annotated using ownership domains, and the typechecker produces a warning.

Annotating the generic `Hashtable` also requires nested parameters: `Hashtable` has three domain parameters for its keys, values and entries. Both `DrawingView` and `DrawingEditor` take `M`, `V`, and `C` as parameters. Although the number of annotations seems excessive and maybe argues in favor of generic ownership (Potanin et al. 2006), the ownership domains for the `Hashtable` key, value and entries need not correspond to the `M`, `V` and `C` ownership domains.

One solution that is not type-safe would be to store the `Hashtable` as `Object`, then cast down to a `Hashtable` upon use. This would be the equivalent of raw types, but without re-implementing them in the ownership domain type system. Another solution would be to refactor the program to eliminate this static field since it gives any object access to all the `DrawingView` and `DrawingEditor` objects. Since eliminating the static field would require a significant refactoring, perhaps another solution would be to support package-level, static ownership domains, similar to confined types (Bokowski and Vitek 1999), or to combine both confinement and own-

```
/**
 * DrawingView renders a Drawing and listens to its changes.
 * It receives user input and delegates it to the current Tool.
 */
interface DrawingView<M,V,C> extends DrawingChangeListener<M,V>... {
  // Add a listener for selection changes
  void addFigureSelectionListener(? FigureSelectionListener<M,V,C> fsl);
  ...
}

class StandardDrawingView implements DrawingView<M,V,C>, ... {

 /**
  * The registered list of listeners for selection changes
  */
 OWNED Vector<? FigureSelectionListener<M,V,C>> fSelectionListeners;

 StandardDrawingView(V DrawingEditor<M,V,C> editor, ...) {
   ...
   // DrawingEditor implements FigureSelectionListener
   // editor is in 'V' domain parameter, not 'C'!
   addFigureSelectionListener(editor);
}

 /**
  * Add a listener for selection changes. AbstractCommand implements
  * FigureSelectionListener. Command is in the 'C' domain parameter!
  */
 void addFigureSelectionListener(? FigureSelectionListener<M,V,C> fsl) {
  fSelectionListeners.add(fsl);
  }
}
```

Figure 4.15: JHotDraw: annotating `addFigureSelectionListener`.

ership in one type system (Potanin 2007).

Observation: Annotations may be unnecessarily verbose. Ownership domain annotations tend to be verbose: e.g., formal method parameters need to be fully annotated even if they are not used in the method body or used in a restricted way. This produces particularly unwieldy annotations for containers of generic types.

In Fig. 4.17, the method `clearStackVerbose()` indicates the current level of annotations needed. It should be possible to leave out domain parameters when they are not really needed. This may involve using implicit existential ownership types as in the method `clearStackAny()`. The question mark annotation could mean that there exists some domain parameters `d1`, `d2`, `d3`, `d4`, such that the formal method parameter `s` could be annotated with `lent<d1<d2,d3,d4>>`. Using appropriate defaults, the annotations could probably be reduced to the level needed to annotate a raw type, as shown in the method `clearStack()`.

```
class NullDrawingView<M,V,C> ... implements DrawingView<M,V,C> {

  static unique Hashtable< ? DrawingEditor<?,?,?>,
                           ? DrawingView<?,?,?>,
                           ?> dvMgr = new ...
     ...

  public synchronized static
     Vx DrawingView<Mx,Vx,Cx>
    getManagedDrawingView<Mx,Vx,Cx> (V1 DrawingEditor<Mx,Vx,Cx> editor) {
      if (dvMgr.containsKey(editor)) {
        Vx DrawingView<Mx,Vx,Cx> drawingView = dvMgr.get(editor);
        return drawingView;
      }
      else {
        Vx DrawingView<Mx,Vx,Cx>newDrawingView = new NullDrawingView(editor);
        dvMgr.put(editor, newDrawingView);
        return newDrawingView;
      }
      ...
  }
}
```

Figure 4.16: JHotDraw: annotating static fields.

```
class UndoManager<M,V,C> {
  /**
   * Collection of undo activities
   */
  OWNED Vector<M Undoable<M,V,C>> undoStack;

  void clearStackVerbose(lent Vector<M Undoable<M,V,C>> s) {
    s.removeAllElements();
  }

  void clearStackAny(lent Vector<? Undoable<?,?,?>> s) {
    s.removeAllElements();
  }

  void clearStack(lent Vector<Undoable> s) {
    s.removeAllElements();
  }
}
```

Figure 4.17: JHotDraw: reducing annotations that are not needed.

Observation: Manifest ownership can reduce the annotation burden. The current defaulting tool annotates `String` objects with `shared`. However, during the annotation process, we found ourselves adding the `shared` annotation to many other types such as `Font`, `FontMetrics`, and `Color`. For example, *manifest ownership* (Clarke 2001), i.e., the ability to specify a global per-type default, rather than an annotation for every instance of a type, could reduce the annota-

tion burden in those cases, and may be worth exploring in future work.

Observation: Reflective code cannot be annotated. JHotDraw uses reflective code to serialize and deserialize its state and such code cannot be annotated using ownership domains (Aldrich et al. 2002c).

Observation: Annotate exceptions as `lent`. We were not particularly interested in reasoning about exceptions, so we annotated exceptions them with `lent`. However, richer annotations are possible, as illustrated by (Werner and Müller 2004).

4.6.1.4 Annotation Summary

The annotations are checked by a type system in a modular fashion, one class at a time. The annotation examples illustrate how a developer adding the annotations mostly provides local hints. In particular, rarely does the developer require missing global information. Of course, some of the harder annotations require computing some reachability, which is perhaps best left for a tool.

4.6.2 Object Graph Extraction

While adding the annotations, I ran the static analysis to extract an object graph based on the annotations, and used the extracted object graphs to visualize the annotations and refine them accordingly. Of course, as long as there are annotation warnings, the object graph may be unsound, but it may still be useful.

During the case study, I made several observations, outlined in bold below. The requirements for a runtime architecture (Section 1.8, Page 22) dictated some of the questions that the observations answer. A taxonomy for software exploration tools by Storey, Müller et al. (Storey et al. 1999), but applied to runtime structures instead of code structures, inspired the others.

Observation: Flat object graphs do not scale. For comparison, I extracted object graphs for JHotDraw using several existing static analyses that have publicly-available tools. For instance, Fig. 4.18 shows the output of WOMBLE (Jackson and Waingold 2001) on JHotDraw. WOMBLE produces a complex, flat object graph where low-level objects such `Dimension` and `Rectangle` appear at the same level as the root application object, `JavaDrawApp`. The PANGAEA output for JHotDraw is even more complex (Fig. 4.19).

Observation: Some object graphs do not correctly reflect aliasing. There are other serious problems with WOMBLE's output. By design, WOMBLE does not handle aliasing soundly. For instance, WOMBLE can shows multiple nodes in the object graph for the same runtime object. In Fig. 4.18, there are multiple `JavaDrawApp` nodes, highlighted in black. Similarly, Fig. 4.18 confusingly shows a separate `DrawingEditor` instance, when it is the same object as the `JavaDrawApp` instance at runtime (`JavaDrawApp` extends `DrawingEditor`).

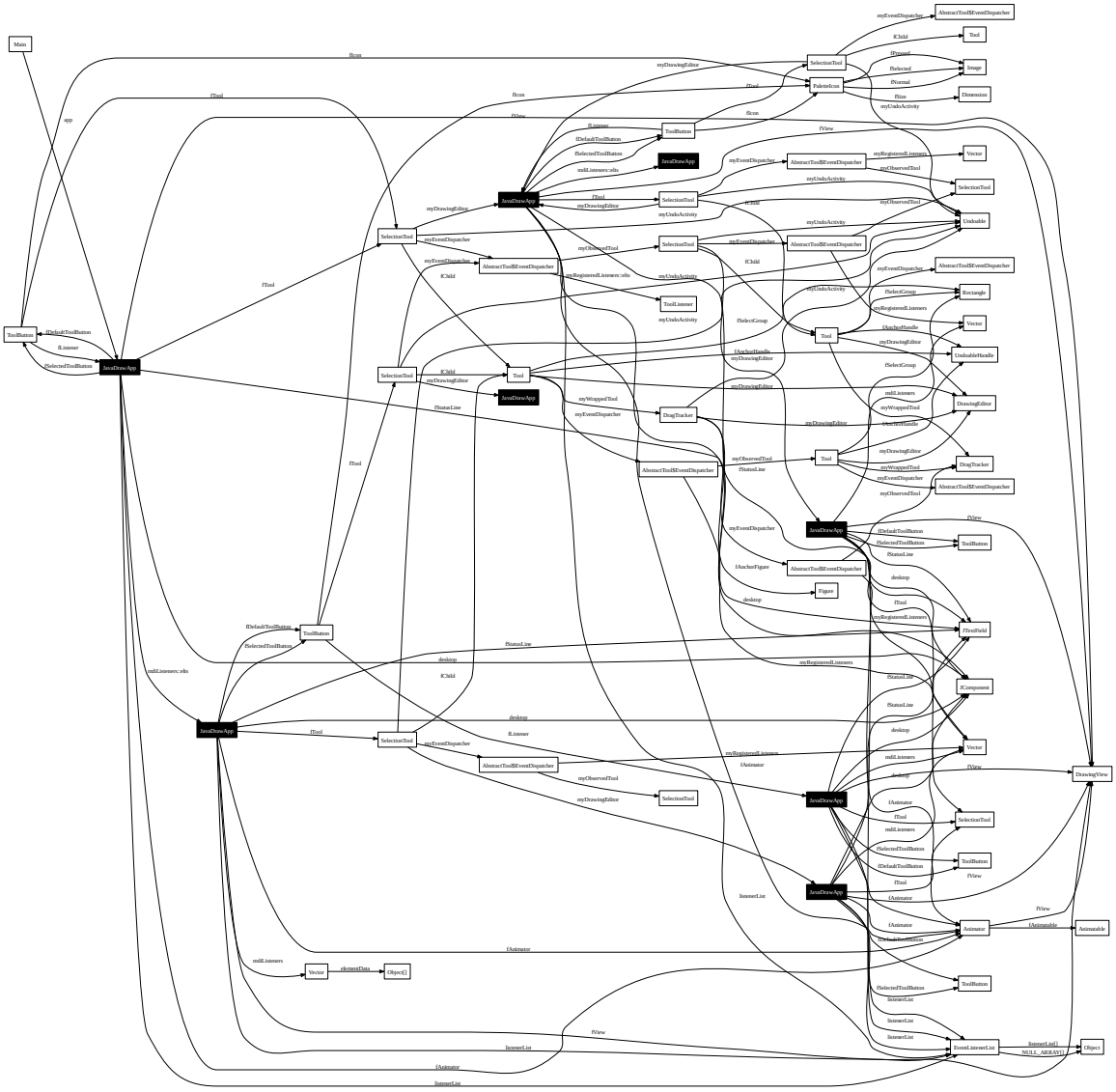


Figure 4.18: JHotDraw: thumbnail of the object graph obtained at compile time by WOMBLE (Jackson and Waingold 2001). The embedded image becomes readable after zooming in by 800%.

Observation: An OOG effectively abstracts objects by ownership hierarchy and by types compared to a non-hierarchical object graph. After adding the annotations, I extracted the OOG in Fig. 4.20. The hierarchical object graph has many fewer objects in the top-level domains compared to the flat object graph, because it collapses lower-level objects underneath other objects.

Collapsing many nodes into one is a classic approach to shrink a graph. However, the OOG statically collapses nodes based on the ownership and type structures, and not according to where objects were declared in the program. Moreover, it is possible to recover the substructure uniformly across all objects by increasing the visible depth of the ownership tree.

In principle, one could manually elide objects in WOMBLE's output (Fig. 4.18) to obtain a

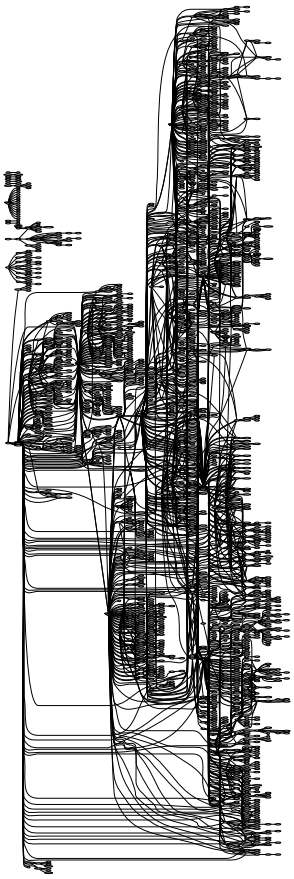


Figure 4.19: JHotDraw: flat object graph for JHotDraw obtained using PANGAEA (Spiegel 2002). The edges correspond to object references. The image is embedded postscript: to obtain a readable diagram, view this document electronically and zoom in by at least 2400%

more abstracted diagram. Indeed, in many architectural extraction approaches, the developer filters elements that satisfy certain query criteria to produce more abstracted views, e.g., by collapsing all the nodes labeled with a common prefix according to some naming convention into a single subsystem (Storey et al. 1999). However, in both cases, the result would still be a non-hierarchical view. Moreover, selecting and eliding from many objects at the same level involves more trial and error. It is also unclear how a developer can decide which objects to elide, and if doing so maintains soundness, i.e., the diagram still shows all objects and relations between them.

Observation: Abstraction by trivial types can unclutter a diagram. However, the default trivial types often leads to imprecision. By default, abstraction by trivial types is turned on. The default list of trivial types includes types such as `Object`, `Cloneable` and `Serializable` from the Java Standard Library. The extracted OOG (Fig. 4.20) is imprecise since I am unable to recognize in it many instances of the core types in the class diagram (Fig. 4.2). Later on, we will refine the list of trivial types to obtain an OOG that conveys more of our architectural intent.

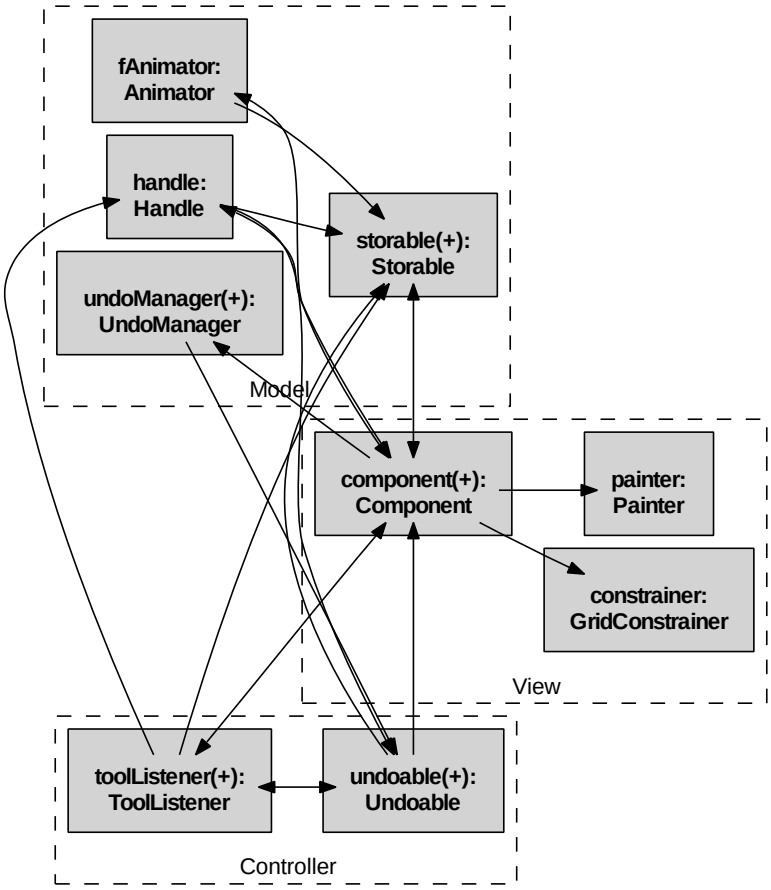


Figure 4.20: JHotDraw: OOG with abstraction by trivial types (the default list).

Observation: Without abstraction by types, an OOG can be very cluttered if there are many related subtypes. Turning off abstraction by types produces an OOG that lacks abstraction (Fig. 4.21). It shows objects for `RedoCommand` and `NewViewCommand`, as well as objects for `ConnectionTool` and `CreationTool`, among others. What we really wanted is to merge all `Command` instances together and all `Tool` instances together, but not merge `Tool` and `Command` instances together.

For example, in JHotDraw, `CommandMenu` declares a `Vector<Command>`. `Vector`'s `ELTS` formal domain is transitively bound to `CONTROLLER`. Recall that `Command` is an interface. For soundness, the analysis creates an edge from the `CommandMenu` object inside `VIEW` to any subtype of `Command` inside `CONTROLLER`, such as `RedoCommand` and `NewViewCommand`. Moreover, a `Command` contains another nested `Command`. So this results in an almost fully connected graph. Because of the large number of top-level objects, this OOG, while hierarchical, is hardly an improvement over a flat object graph such as the one WOMBLE obtains from a bytecode program, without relying on annotations (Fig. 4.18). Thus, abstraction by ownership hierarchy is insufficient, and additional abstraction is needed to reduce the number of objects compared to a flat object graph.

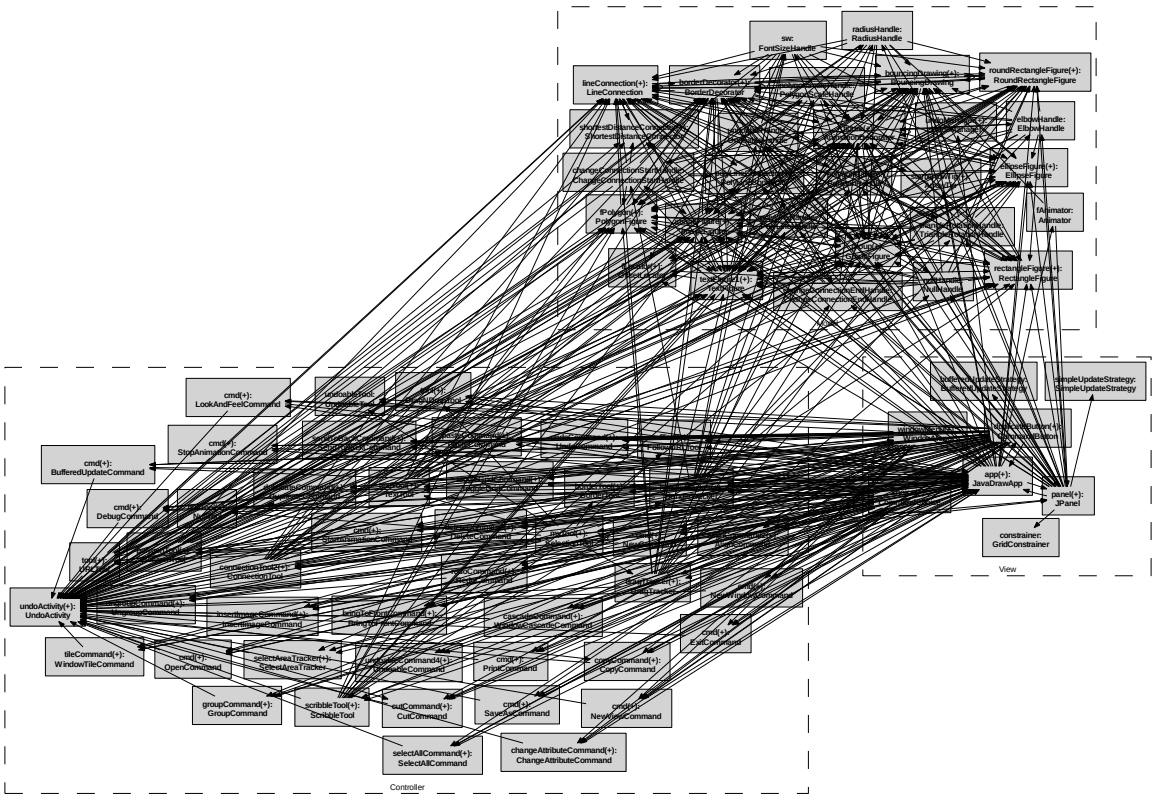


Figure 4.21: JHotDraw: thumbnail of the OOG based on an instantiation-based view, but without abstraction by types. The embedded image becomes readable after zooming in by 800%.

Observation: With carefully chosen trivial types, an OOG effectively abstracts related instances. I turned on abstraction by trivial types, initially using the default list of trivial types, which produced an OOG where each display object merges too many field declarations (Fig. 4.20).

ArchRecJ assists a developer in selecting non-default trivial types as follows. First, the developer graphically selects an object which appears to merge too many objects. The tool then displays an inheritance hierarchy of the types of the field declarations that the selected object merges. The general principle is that the developer must select a type that would cut the path from an interesting leaf type in the inheritance hierarchy up to an uninteresting common ancestor (Fig. 4.22).

I followed the above process to select the trivial types for JHotDraw. JHotDraw has its own list of interfaces that many classes implement such as `Storable` and `Animatable`, which I proceeded to add to the list of trivial types. I also added several constant interfaces such as `SwingConstants`[5].

In addition, many types in JHotDraw extend or implement listener interfaces to realize the Observer design pattern. For instance, both interfaces `Command` and `Tool` are in `CONTROLLER` and both extend the interface `ViewChangeListener`. I also added many of the listener interfaces as

[5]Inheriting from a constant interface is a bad coding practice, the Constant Interface *antipattern* (Bloch 2001, Item #17), and Java 1.5 supports *static imports* to avoid it. This is one more reason to avoid that practice.

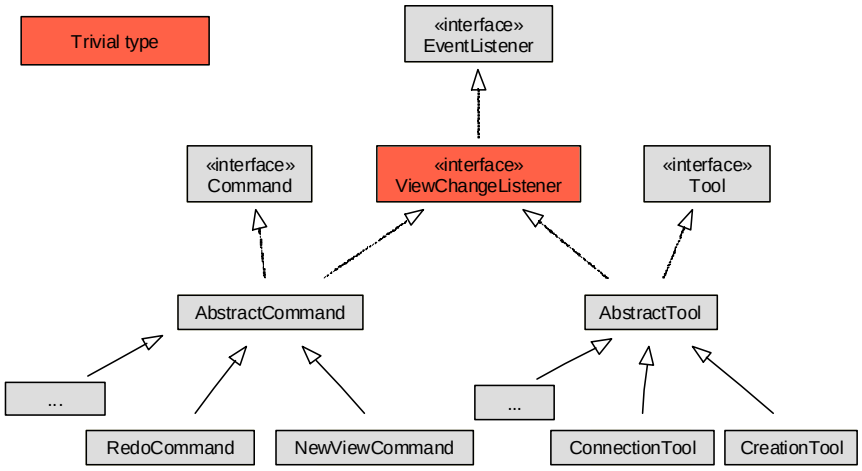


Figure 4.22: JHotDraw: making `ViewChangeListener` a trivial type.

trivial types.

With the refined list, the analysis merges `RedoCommand` and `NewViewCommand`, because `Command` is their non-trivial LUB. Similarly, it merges `ConnectionTool` and `CreationTool`. But the analysis does not merge `ConnectionTool` and `RedoCommand` because their LUB, `ViewChangeListener`, is a trivial type (Fig. 4.22). Thus, using the non-default trivial types provides a more meaningful OOG (Fig. 4.23). In that OOG, we recognize separate `Tool` and `Command` objects in `CONTROLLER`. Similarly, `MODEL` shows distinct `Figure`, `Handle` and `Connector` objects, all architecturally significant.

Because of JHotDraw's complex inheritance hierarchy, I had to fine-tune the list of trivial types to achieve the desired level of abstraction—more so than for the other subject systems. For example, another subject systems I analyzed, Aphyds (Section 7.5), did not require using abstraction by types.

Riehle previously studied JHotDraw and produced manually a code architecture (Fig. 4.2). Riehle posited that the original JHotDraw designers used the following techniques to present the JHotDraw design in their tutorials: (a) *merge interface and abstract implementation class*, because such a code factoring, although important for code reuse, is often unimportant from a design standpoint; and (b) *subsume a set of similar classes under a smaller set of representative classes*, because showing many similar subclasses that vary only in minor aspects often leads to needless clutter (Riehle 2000, pp. 139–140).

The OOG achieves results similar to the above heuristics. For instance, all runtime `Handle` objects referenced in the program by the `Handle` interface, its abstract implementation class `AbstractHandle`, and any of its concrete subclasses such as `ElbowHandle` or `NullHandle`, appear as one `Handle` display object in the `MODEL` tier. An OOG can sometimes suffer from a precision loss: not all `Handle` classes have a field reference to a `Locator` as Fig. 4.2 indicates. Only `NullHandle` and its subclasses do. But since they were all merged into `Handle`, the OOG shows an edge from `Handle` to `Locator` in Fig. 4.25.

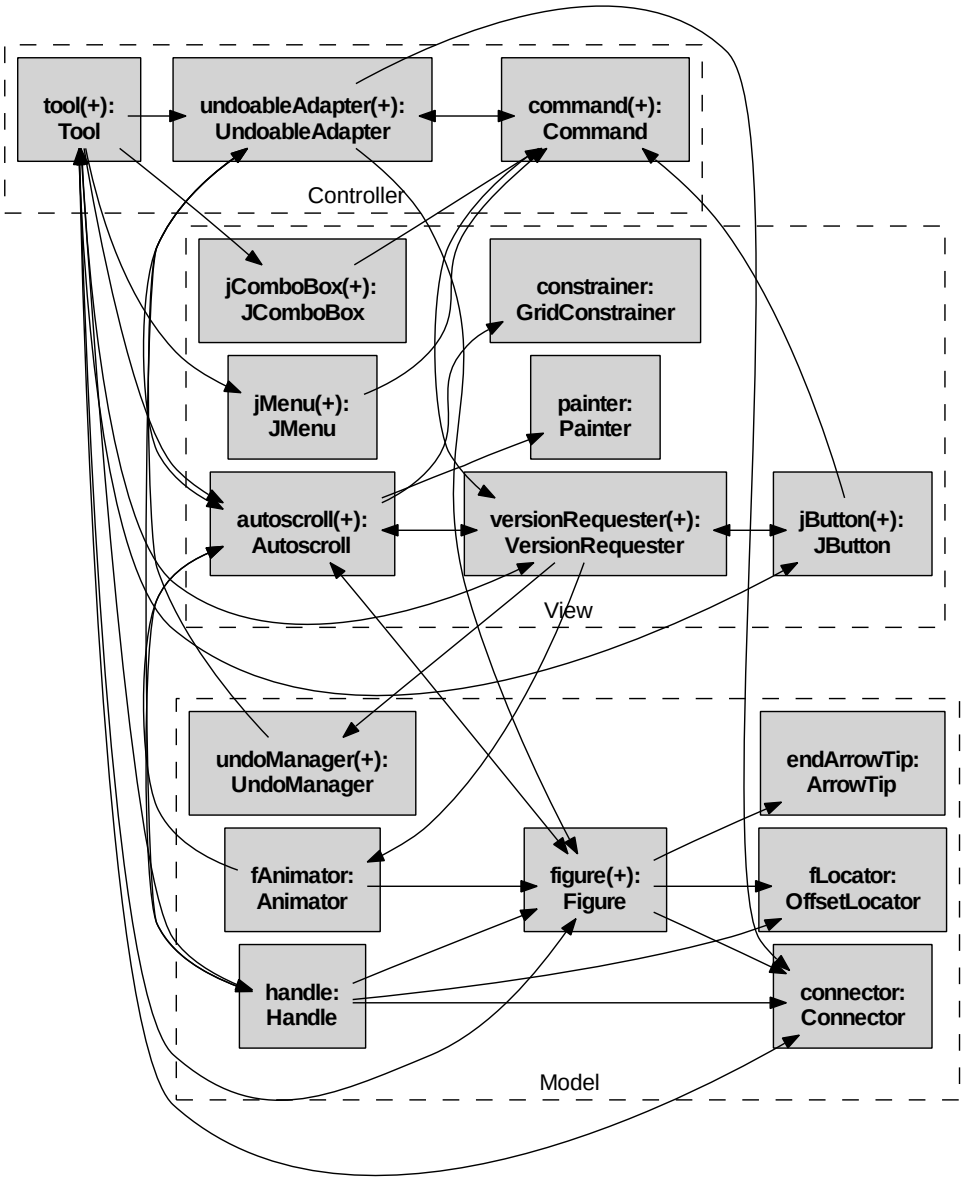


Figure 4.23: JHotDraw: OOG with abstraction by trivial types (the fine-tuned list).

Observation: Abstraction by types can help identify unexpected subtyping relationships in the program, some of which could point to design problems. With abstraction by trivial types turned on, I was surprised that the OOG did not show any instances of the `Figure` type, presumably one of the core types in the class diagram. I used ArchRecJ to obtain the field declarations that a display object merges (See Fig. 4.1) and used that information to determine that one object, `textFigure1:Drawing`, merged objects of type `Figure` and `Drawing` in the `MODEL` domain.

I traced these field declarations to the code, and discovered by code inspection that indeed, 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

the Release Notes for JHotDraw Version 5.1 mentioned this fact, it was still unexpected. In the `framework` package, interface `Drawing` does not extend `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" (Gamma 1998, Slide #16).

I was slightly surprised when I inadvertently added interface `Handle` as a trivial type. This resulted in an OOG with one object for `NullHandle` (which directly implements `Handle`) and another object for all instances of the concrete subclasses that implement `Handle` by extending `AbstractHandle`. While this result seemed counter-intuitive, that OOG was still sound: there is no runtime object that can have both types `NullHandle` and `AbstractHandle`, so no one runtime object appears as two display objects in the OOG.

Observation: Abstraction by design intent types can achieve higher precision than abstraction by trivial types. Abstraction by trivial types can quickly unclutter an OOG, but is not very precise. For instance, the JHotDraw OOG based on trivial types does not show distinct `Drawing` and `Figure` objects (Fig. 4.25). Presumably, both interfaces are architecturally relevant. This is because the base class that implements `Drawing`, `StandardDrawing`, extends `CompositeFigure`, which in turn implements `Figure`. But `Drawing` does not extend `Figure` and is not a trivial type. Merging objects based on non-trivial LUBs, coupled with merging objects after the fact for soundness, causes field declarations of type `Drawing` and `Figure` to get merged in `MODEL`. An object may have multiple types, but some types may be more architecturally relevant than others. In this example, `StandardDrawing` extends `CompositeFigure` to enable nesting a `Drawing` inside another `Drawing`. In this case, we would like to view a `StandardDrawing` object as a `Drawing` object, instead of a `Figure` object.

JHotDraw's `framework` package includes abstract classes and interfaces that define the core framework. I added to the list of design intent types all the types in the `framework` package and ordered them from most to least architecturally relevant, e.g., `Drawing` appears before `Figure`.

When deciding whether to merge two field declarations `StandardDrawing` and `CompositeFigure`, the analysis finds the design intent type `Drawing` in the list, since `StandardDrawing` is a subtype of `Drawing`. Similarly, it finds the type `Figure`, since `CompositeFigure` is a subtype of `Figure`. Because `Drawing` is not a subtype of `Figure`, the analysis does not merge objects `StandardDrawing` and `CompositeFigure`. But it does merge `StandardDrawing` and `BouncingDrawing`. Similarly, it merges `EllipseFigure`, `RectangleFigure`, etc. But it keeps objects of type `Drawing` and `Figure` distinct in `MODEL` (Fig. 4.24), just as we desired.

Observation: An OOG provides architectural abstraction by showing architecturally significant objects near the top of the hierarchy and data structures further down. A key issue in architectural extraction is distinguishing between objects that are architecturally relevant and those that are not. The OOG provides architectural abstraction by pushing lower-level objects underneath higher-level objects. As a result, the OOG does not show non-architecturally relevant objects in the top-level domains.

An OOG shows objects inside domains, and provides an instance granularity larger than an object. For instance, the `CONTROLLER` tier includes `Command` and `Tool` instances, rather than a

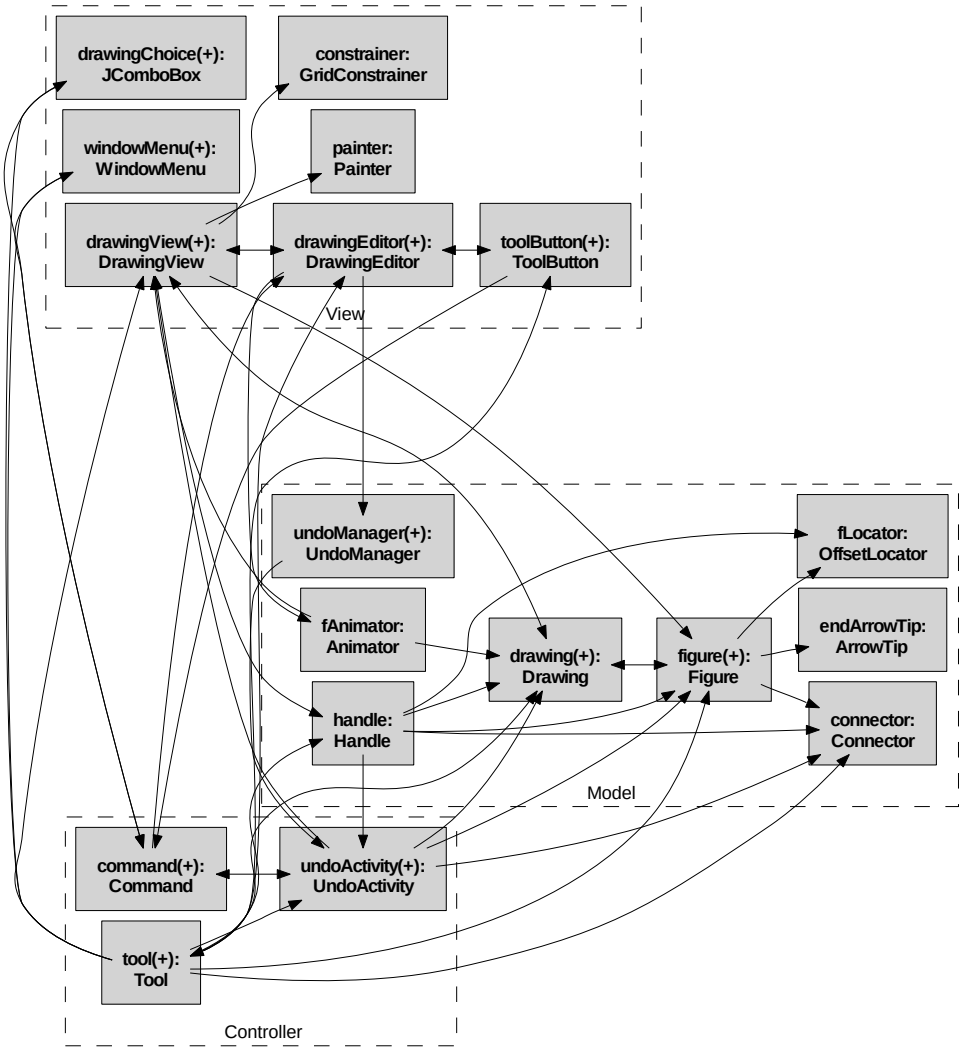


Figure 4.24: JHotDraw: OOG with abstraction by design intent types.

Controller component. In contrast, the `VIEW` domain also has a `DrawingView` object.

There are three top-level domains: `MODEL`, `VIEW` and `CONTROLLER`. The OOG in Fig. 4.25 seems to have the right level of abstraction since we recognize in it most of the core types from the class diagram (Fig. 4.2).

A rule of thumb in architectural documentation is to have 5 to 7 components per tier (Koning et al. 2002). Thus, the number of objects in each domain is similar to the number of components in tiers found in typical architectural diagrams: `MODEL` has 14 objects, `VIEW` has 6 objects, and `CONTROLLER` has 3 objects.

One could split the `MODEL` domain into one domain for *application model* objects, such as instances of `UndoManager` and `StorageFormatManager`, and one for *domain model* objects, with `Figure`, `Handle` and related objects, as in the Model-Model-View-Controller pattern[6].

The OOG (Fig. 4.25) has only 23 objects in the top-level domains. In contrast, existing

[6] `http://c2.com/cgi/wiki?ModelModelViewController`

compile time object graph analyses that do not rely on annotations produce flat object graphs that show all objects at the same level, e.g., the `Dimension` and `JavaDrawApp` objects in Fig. 4.18.

In JHotDraw, `Point` objects are immutable, so we annotated them with `unique` to pass them linearly, as discussed in Section 4.4.2. Hence, they do not appear in the OOG.

Observation: Hierarchy allows showing both the high-level structure of the object graph and the low-level details at various levels of abstraction. Ideally, an architectural diagram "can be read in 30 seconds, in 3 minutes, and in 30 minutes" (Koning et al. 2002). For example, Fig. 4.25 can be considered a 30-minute OOG.

There are two ways to control the level of detail. One is to control the unfolding depth of the DisplayGraph, which affects the depth of the object substructures uniformly for all objects starting from the root object. Because one object's substructure may be more interesting than that of some other object, ArchRecJ allows the developer to collapse the internals of a selected object; in that case, the tool appends the $(+)$ symbol to that object's label. In Fig. 4.25, we manually elided the substructure of all the objects in the top-level domains except for `Drawing`, to highlight the Composite pattern. Inside `Drawing`, the `OWNED` domain shows several objects. We recognize a `Vector<Figure>`, `fFigures`, that maintain the list of sub-figures, and a lifted edge from `fFigures` to `figure:Figure` in `MODEL`. We chose to show `QuadTree`'s substructure, but elided `FigureAttributes`'s substructure.

A 30-second OOG shows the three top-level ownership domains, `MODEL`, `VIEW` and `CONTROLLER` (Fig. 4.26). In addition, dotted edges summarize the field reference edges between objects inside those domains. This high-level overview shows how objects in `MODEL` refer to objects in `VIEW` to send them change notifications. `VIEW` objects have references to `MODEL` objects to display them. Similarly, `VIEW` objects have references to `CONTROLLER` objects. `CONTROLLER` has references to `MODEL` and `VIEW`, but `MODEL` has no references to `CONTROLLER`.

Observation: The OOG is extracted quickly and iteratively refined. Examining the extracted OOGs helped us refine the annotations. For instance, we initially placed `Handle` instances in the `CONTROLLER` domain, but later moved them to the `MODEL` domain, since `Handle` is related to `Figure`.

Assuming ownership annotations are already present, ArchRecJ can extract an object graph with minimal end-user interaction. The user can optionally abstract the object graph by controlling the abstraction by types.

ArchCheckJ and ArchRecJ are sufficiently fast to allow a developer to iteratively refine the extracted object graph. Computing the OOG in Fig. 4.25 takes less than 20 seconds on a modest Intel Pentium 4 (3 GHz) with 2 GB of memory.

Observation: An OOG shows potentially useful information about the system's runtime structure. One could point to several useful pieces of information in the JHotDraw OOG.

- **System decomposition:** Decomposition information is often useful to have. In the OOG, 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.

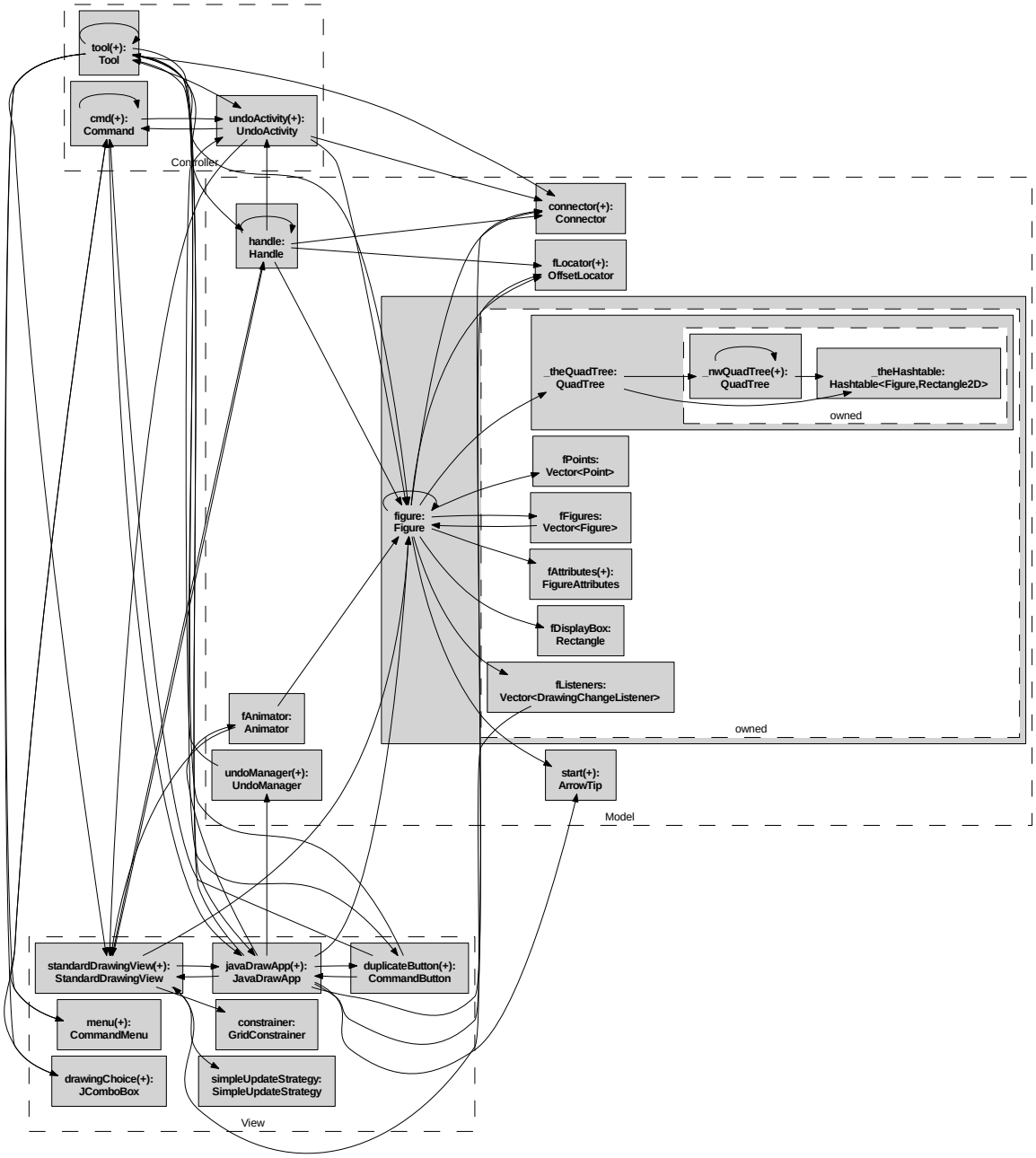


Figure 4.25: JHotDraw: top-level OOG. The objects in the top-level domains are collapsed, except for the object labeled `figure:Figure`.

For example, `Drawing` is a `CompositeFigure`. Following the Composite pattern, it maintains a list of its sub-figures. Indeed, viewing the decomposition of `textFigure1` reveals, among others, an object `fFigures` of type `Vector<Figure>`, inside its `OWNED` domain. When performing system decomposition, the inside of a component is related to its outside. Indeed, there is a lifted edge from `fFigures` to `textFigure1` in the `MODEL` – since `textFigure1` merges both `Figure` and `Drawing`.

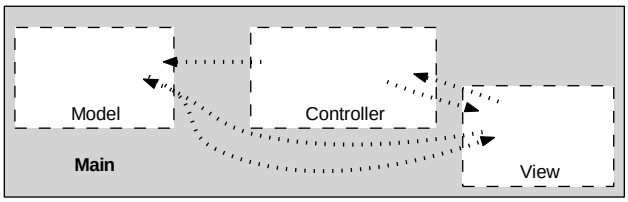


Figure 4.26: JHotDraw: Model-View-Controller summary. The dotted edges summarize field reference edges between objects in the top-level domains.

The `DrawingView` interface extends the `DrawingChangeListener` interface. Hence, the OOG shows an edge from object `fListeners` inside object `Figure` to the `DrawingView` object. Inside object `Figure`, object `fFigures` contains the composite `Figure` objects.

- **Object encapsulation:** To highlight the cases of strict encapsulation, the OOG uses a thick dashed border for a private domain that is not linked to a parameter. For instance, in Fig. 4.25, the `Map` object is encapsulated inside the `OWNED` domain of the `FigureAttributes` object.
- **Object references:** The OOG indicates the presence or absence of field references between objects. The OOG highlights for instance how, in the MVC pattern, a view redraws itself when the model notifies it of state changes. The core model object, `Drawing`, maintains `fListeners`, a list of `DrawingChangeListener` objects, that are notified whenever the `Drawing` changes. Interface `DrawingView` extends `DrawingChangeListener`, hence the edge from `fListeners` to `myDrawingView`. A `Tool` object merges instances of type `Tool` and `UndoableTool`. An `UndoableTool` is a wrapper object around a `Tool` object. This explains the self-edge on `Tool` in Fig. 4.25.
  We were surprised by the lack of field references from the `MODEL` to the `CONTROLLER` in Fig. 4.26. In the base MVC pattern, a controller registers itself with the model to receive notifications. Our explanation is that JHotDraw follows the MVC pattern, but slightly modified in two ways. First, the Command Processor pattern (Buschmann et al. 1996, p. 277) is used to address the "close coupling of views and controllers to a model" in the base MVC pattern (Buschmann et al. 1996, p. 142). Second, a `DrawingView` acts as both a view and a controller. This is a common optimization in the MVC pattern since the view and the controller are tightly coupled. Indeed, in the JHotDraw "CRC Cards View", the designers mention that `DrawingView` "handles input events" (Gamma 1998, Slide #10), which is a typical controller responsibility.
- **Object soundness**: while demonstrating soundness requires a formal proof, we visually inspected the OOGs to test the implementation of the ArchRecJ tool. For example, the OOG shows only one canonical object to represent the application object, `app:DrawingEditor` (Fig. 4.25), unlike WOMBLE's output (Fig. 4.18), which shows two `JavaDrawApp` and `DrawingEditor` distinct objects.

Observation: A tool can enforce structural constraints on the OOG. We think the OOG, together with effective change management, can help prevent architectural drift or erosion during software evolution, more effectively than the program, with or without annotations. In the unan-

notated program, changing the runtime structure is as simple as passing a reference to an object. The ownership annotations help somewhat. But a developer can still add communication paths by adding domain links, declaring additional domain parameters and passing additional domain arguments at object allocation sites. Code reviews could audit such changes.

If the OOG reflects such architecture-modifying changes, the OOG makes it easier to trigger an architecture review. A visual inspection of the OOG could look for suspected architectural violations. Or once the OOG is converted to a C&C view in an ADL, the ADL can enforce global constraints on the runtime structure (Section 7.8.9).

Using ownership domain annotations to enforce constraints may require code changes. 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 (Section 4.6.1). The OOG can enforce such a constraint without requiring changing the annotations or the code. In addition, domain links treat all communication equally, forcing developers to add domain links. But a policy can 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" (Buschmann et al. 1996, p. 127).

4.6.3 JHotDraw Summary

JHotDraw has a complex inheritance hierarchy and implements many design patterns. However, I was able to add annotations to it, and extract hierarchical object graphs that convey more architectural abstraction than any of the previous flat object graphs.

4.7 Extended Example: HillClimber

By many accounts, JHotDraw is the brainchild of object-oriented analysis and design (OOAD) experts. In the next case study, I evaluated using the annotations and the static analysis on a subject system that OOAD novices designed.

4.7.1 About HillClimber

The second subject system, HillClimber, is a 15,000 line Java application that was developed by undergraduates at the University of British Columbia (UBC). HillClimber is part of a collection of Java applications to graphically demonstrate artificial intelligence algorithms, built on the `CIspace` framework (Poole and Macworth 2001). In particular, HillClimber, demonstrates stochastic local search algorithms for constraint satisfaction problems. HillClimber is also interesting because it uses a framework and its architectural structure had degraded over the years.

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*.

I extracted a UML class diagram from the HillClimber implementation using Eclipse UML (Omondo 2006) (Fig. 4.27).

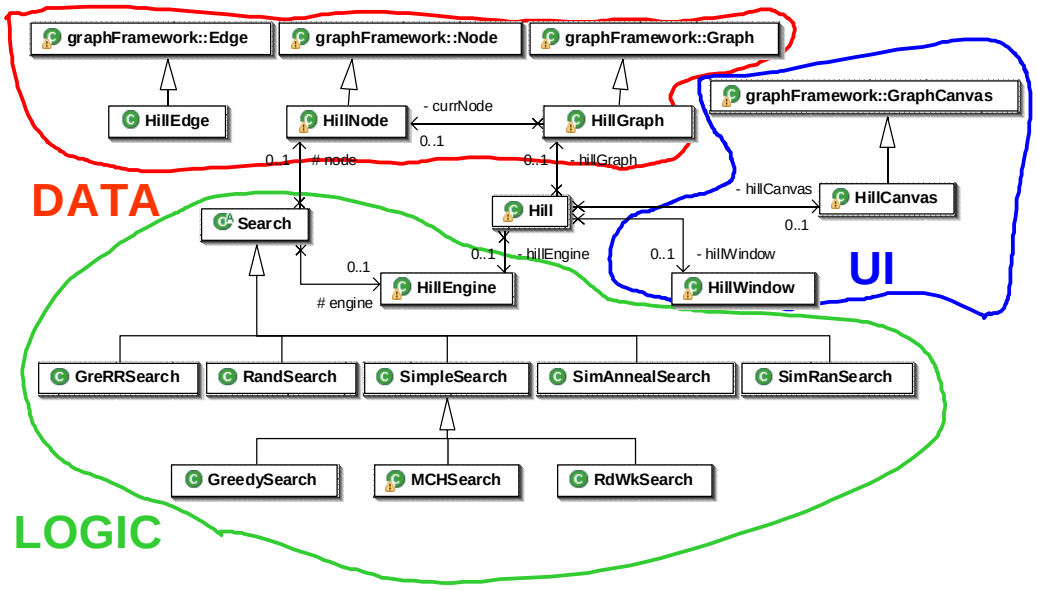


Figure 4.27: HillClimber: partial UML class diagram obtained from the original implementation using Eclipse UML (Omondo 2006). This diagram does not reflect some types introduced during refactoring, such as `IGraph`, `IHillGraph` and `ICanvasMediator`.

4.7.2 Annotation Process

In this section, I briefly discuss the annotation process for HillClimber.

4.7.2.1 Annotation Overview

I organized the HillClimber objects into the following domains:

- `DATA`: stores the graph objects, namely instances of `Graph`, `Node`, etc., and those of their subclasses, `HillGraph`, `HillNode`, etc.;
- `UI`: holds user interface objects;
- `LOGIC`: holds instances of `HillEngine`, `Search` and subclasses thereof, and associated objects.

While adding annotations to HillClimber, I refactored the code to reduce the coupling between some of the objects the `UI` and `DATA` domains, as I discuss below.

4.7.2.2 Annotation Examples

Observation: Ownership domains expose implicit communication. In HillClimber, adding ownership domain annotations exposed covert object communication through base classes from two parallel inheritance hierarchies. During an early iteration, we parameterized the base class `GraphCanvas` by the `UI` and `DATA` domain parameters. We then realized that `Graph`, the base class for `HillGraph`, required the `UI` domain parameter (Fig. 4.28). Class `Graph` needed the `UI` domain parameter only to properly annotate a `GraphCanvas` field reference, which we did not expect. In turn, this revealed that `HillGraph` and `HillCanvas` were communicating through

```
/******** Before programming to interface **********/
class HillNode<UI,LOGIC,DATA> extends Node<DATA> {
   DATA HillGraph<UI,LOGIC,DATA> hillGraph;
}

/******** After programming to interface **********/
class HillGraph<UI,LOGIC,DATA> extends Graph<DATA>
           implements IHillGraph<DATA> {
}

interface IHillGraph<DATA> extends IGraph<DATA> {
}

class HillNode<DATA> extends Node<DATA> {
   DATA IHillGraph<DATA> hillGraph;
}
```

Figure 4.28: HillClimber: refactoring `HillGraph` to program to an interface.

their base classes `Graph` and `GraphCanvas`. In the end, I moved the reference to `GraphCanvas` from `Graph` to `HillGraph` and generalized it as an `IHillCanvas` reference by extracting an interface `IHillGraph` from `HillGraph`. As a result, the class `Graph` no longer needed the `UI` domain parameter.

Observation: Ownership domain annotations highlight tight coupling and promote decoupling code. Ownership domain annotations programming practices that decouple code, such as programming to an interface, or using the mediator pattern, as we discuss below.

Programming to an Interface. It is recommended to "refer to objects by their interfaces" (Bloch 2001, Item #34) since interfaces can reduce coupling between classes by splitting intent from implementation. When adding annotations to an interface requires fewer domain parameters than annotating the corresponding class, the annotations can enforce this idiom. In particular, an implementation class can require a private ownership domain to be passed as an actual value for one its parameters. Since a private ownership domain cannot be named by an outside client, the client is then forced to use the interface which does not require these parameters.

For HillClimber, we used the technique of hiding the extra ownership domain parameter behind an interface, to force a client to access an object only through the interface—the client may not even cast the interface reference to an implementation class.

The original implementation for class `HillNode` had a field reference of type `HillGraph`. However, `HillGraph` took the three domain parameters `UI`, `LOGIC` and `DATA`, which required passing all those parameters to `HillNode` (Fig. 4.28).

This demonstrates that encountering an unexpected domain parameter while adding the annotations often indicates unnecessary coupling. For instance, why should `HillNode` require the `UI` domain parameter? Thus a lengthy domain parameter list can be an objective measure of a code smell (Abi-Antoun et al. 2007a). Furthermore, ownership domain annotations can help a developer lower the coupling by suggesting which specific type declarations need to be generalized to shorten the list of domain parameters on the enclosing type.

```
1  abstract class Entity<DATA> {
2    DATA Graph<DATA> graph; // parent graph
3    ...
4  }
5
6  class Node<DATA> extends Entity<DATA> {
7   ...
8   int getHeight() {
9      return graph.getCanvas().getFontMetrics()...;
10   }
11 }
```

Figure 4.29: HillClimber: before using a mediator.

In HillClimber, one solution was to extract an `IHillGraph` interface from class `HillGraph` that requires only the `DATA` domain parameter and make a `HillNode` object reference the `HillGraph` object through the `IHillGraph` interface. We decided against carrying this refactoring further and eliminating the `UI` and `LOGIC` domain parameters on `HillGraph` itself.

Since `HillGraph`, `HillNode`, etc., form a parallel inheritance hierarchy to `Graph`, `Node`, etc., respectively, a similar refactoring was performed on `Graph` by extracting a `IGraph` interface—although `Graph` and `IGraph` both take the domain parameter `DATA` (Fig. 4.28), so programming to an interface would not hide any domain parameter.

We observed tightly coupled code throughout HillClimber. Similarly, we were surprised that a dialog class `FontDialog` required the `DATA` domain parameter. It turned out that `FontDialog` had a field reference declared with its most specific type `GraphCanvas`. In some cases, it is possible to generalize the type of the reference, e.g., use `java.awt.Frame` to eliminate the need for the domain parameter. However, `FontDialog` needed access to some of the `GraphCanvas` functionality, so this required a different solution, namely, using a mediator, as I discuss below.

Mediator Pattern. Defining an interface is sometimes insufficient to decouple code since referring to an object through its interface still requires access to the domain the object is in. One solution is to use the Mediator design pattern (Gamma et al. 1994, p. 273), as shown here.

In the original HillClimber implementation (Fig. 4.29), a `Node` obtained a reference to a `GraphCanvas`, and this violates the Law of Demeter (Lieberherr and Holland 1989) which states that objects should talk only to their immediate neighbors.

Extracting an `IGraphCanvas` interface from `GraphCanvas` would not work, as the `IGraphCanvas` reference would still need to be annotated with `UI`, which is not in scope or a domain parameter. Moreover, the implementation of `getFontMetrics()` could not be moved to `Graph` as it required access to objects in the `UI` domain (Fig. 4.30).

Instead, I defined a mediator (Fig. 4.31). `GraphCanvas` initializes the mediator, and `Entity` and `Node` can then use the mediator (Fig. 4.32).

4.7.3 Object Graph Extraction

I used the extracted object graph to fine-tune the ownership domain annotations in the program and reduce the number of objects in the top-level domains (Fig. 4.33), using the strategies discussed in Section 4.4.2. Using HillClimber, we reconfirmed many of the previous observations.

```
interface IGraphCanvas {
}
// Hide domain parameter UI behind interface
class GraphCanvas<UI,DATA> implements IGraphCanvas {
}

abstract class Entity<DATA> {
  UI IGraphCanvas canvas; // UI unbound
  ...
}

class Node<DATA> extends Entity<DATA> {
 ...
 int getHeight() {
   return canvas.getFontMetrics()...;
  }
}
```

Figure 4.30: HillClimber: extracting an interface (bad attempt).

```
/**
 * Mediator interface
 */
interface ICanvasMediator {
  shared FontMetrics getFontMetrics();
}

/**
 * Mediator implementation class
 */
class CanvasMediatorImpl<UI,DATA> implements ICanvasMediator {
  UI GraphCanvas<UI,DATA> canvas = null;

  CanvasMediatorImpl(UI GraphCanvas<UI,DATA> canvas) {
   this.canvas = canvas;
  }

  shared FontMetrics getFontMetrics() {
      return this.canvas.getFontMetrics();
  }
}
```

Figure 4.31: HillClimber: defining a mediator.

Observation: In practice, there are several opportunities to use strict encapsulation to reduce the clutter. We reduced the clutter in the `DATA` domain by pushing more objects into private domains of other objects. For instance, we placed `heap:HillHeap` inside a private domain of `graph:HillGraph`. We also pushed several `Vectors` into private domains and ensured that the other references to them were `unique` (they were actually passed linearly between objects). In a few cases, we changed the code to prevent representation exposure by returning a copy of an internal list instead of an alias.

```
class GraphCanvas<UI,DATA> extends ... {
 DATA CanvasMediatorImpl<UI,DATA> mediator;
 ...
 DATA ICanvasMediator getMediator() {
   return mediator;
 }
}

abstract class Entity<DATA> {
  DATA ICanvasMediator mediator;
  ...
}

class Node<DATA> extends Entity<DATA> {
 ...
  /**
   * Gets the height of this node.
   *
   */
  protected int getHeight() {
    return mediator.getFontMetrics().getHeight() + ...;
  }
}
```

Figure 4.32: HillClimber: using a mediator.

Observation: In practice, there are several opportunities to use logical containment to reduce the clutter. We defined public domains to reduce the number of top-level objects. A public domain can group related objects, by pushing the contained objects down the ownership tree and removing them from the top-level domains, while keeping those inner objects accessible to objects that can access the outer objects. For example, object `search` has a `HEURISTICS` public domain with two array objects inside it; its peer object `heuristics` inside `LOGIC` accesses those array objects directly[7].

As an aside, I could have used a static analysis to infer the better `OWNED` and `unique` annotations, e.g., (Liu and Milanova 2007; Ma and Foster 2007). But today's annotation inference algorithms cannot infer meaningful domain parameters or public domains (Aldrich et al. 2002c).

Observation: An OOG can provide meaningful architectural abstraction. The HillClimber OOG (Fig. 4.33) shows clearly the core top-level objects, `window`, `canvas`, `engine` and `graph`. Similarly, the `Search` object in the `LOGIC` domain merges many instances of several sub-classes of the class `Search` such as `MCHSearch`, `RandSearch`, etc.

I had introduced `CanvasMediator` during a refactoring to decouple the code. The `window` object merges several user interface objects such as dialogs, and illustrates abstraction by types.

[7]Such an object relation would be prohibited by an owner-as-dominator type system, e.g., (Clarke et al. 1998). This is one case which illustrates the need for the additional expressiveness of logical containment using public domains in the ownership domain type system.

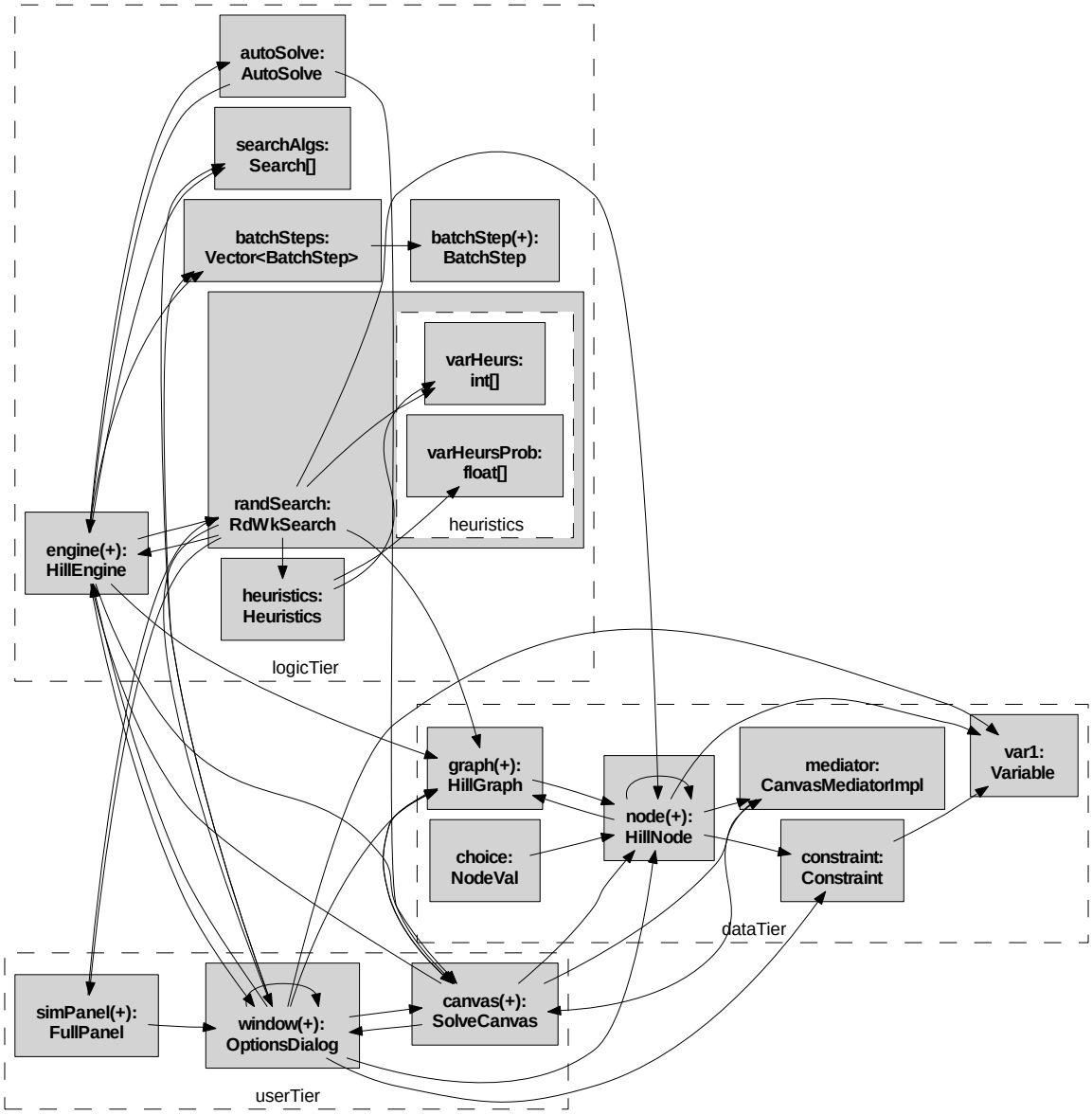


Figure 4.33: HillClimber: top-level OOG.

4.7.4 HillClimber Summary

The HillClimber system is not as well-designed as JHotDraw. Still, I was able to add annotations, run the static analysis, and extract OOGs that provide meaningful architectural abstraction and have sufficient precision.

4.8 Field Study: LbGrid[8]

As a research method, a field study can evaluate how well a software tool or method works with real code and users (Kitchenham et al. 1995).

4.8.1 Overview

The case studies I conducted on the previously described object-oriented systems assessed both the usability of the technique and the engineering tradeoffs that it entails, and led to a more comprehensive week-long on-site field study with an industrial partner. During the field study, we extracted the object graph of a 30-KLOC portion of a large 250-KLOC Java system.

At a high-level, the field study involved selecting a target portion of the system, communicating with the original developers of the code to understand their design intent, adding annotations to the code, typechecking the annotations, running the static analysis to extract an object graph, showing snapshots to the developers, and incorporating their feedback, by refining the annotations and the extracted object graphs.

4.8.2 Research Questions

I refer to the person who conducted the field study, i.e., myself, as the *experimenter*. The *developer* is the person who was familiar with the code being analyzed.

In addition to the earlier questions (Section 4.2, Page 121), we wanted the field study to help answer the following research questions:

- Will an outside developer understand abstraction by ownership hierarchy and by types?
- *How much* effort will it take? *How long* before one can obtain initial architectural diagrams?
- Can one add annotations for the top-level object graph, then extend those annotations down to the rest of the system?
- Can we meaningfully analyze only a part of a system?
- Can we evaluate qualitatively the precision of the analysis by having a developer visually examine the output OOG? For instance, does the OOG omit objects that the developer expected to see? Or does the developer not recognize some of the objects that show up in an OOG?
- How can we improve the usability of the tools?

4.8.3 Setup and Methodology

Pilot constraints. The SCHOLIA tools are plugins in the Eclipse Java development environment. So, in terms of selecting the subject system, we required a module that is Java-based. Since we were adding the annotations manually, we required a module under 50 KLOC in size. In some of the earlier evaluations, e.g., HillClimber (Section 4.7), we refactored the subject system while adding the annotations. During the field study, we wanted to extract the *as-is* object

[8] Portions of this section appeared in (Abi-Antoun and Aldrich 2008b)

graph. We also did not want to explain the annotations or the static analysis to the developers, nor did we expect to involve them with the tools. The developers would be free to refactor based on any insights they gained from the extracted architecture.

The plan. Architectural extraction typically starts by gathering or eliciting documentation from developers who are familiar with the code. Ideally, a developer would document the designed or target runtime architecture, but realistically, we knew that we may have to settle for a class diagram.

Data collection. The experimenter measured the effort by keeping track of the different activities in a time log, and measured the end-to-end time, minus interruptions. He also kept a log of annotation cases that revealed facts about the code such as representation exposure or tight coupling.

The experimenter kept track of the iterations, and what he changed between iterations, such as changing the settings or inputs to the tools. He saved intermediate snapshots of the extracted object graph. He also wrote detailed notes to simulate the thinkaloud protocol (he could not actually speak as he was sitting with others in an open-floor space). After the study, we used the Eclipse history data for each file to analyze how the annotations evolved.

Subject selection. The experimenter ran the jMetra (hyperCision Inc. 2008) code measurement tool on the Java code base, and identified a module of around 30KLOC, excluding unit test code, which we refer to as LbGrid. LbGrid is a multi-dimensional feature-rich grid control that consists of around 300 classes (jMetra includes only static inner classes in the class count, and LbGrid uses non-static inner classes extensively).

In previous evaluations, we used code bases developed prior to Java 1.5 and refactored them to use generics to improve the precision of the analysis. In this case, the code already used generic types. As a bonus, a developer who was familiar with that module would be available.

Static analysis. At no time during the field study did the experimenter run the system. That would have required setting up a complex client-server system, and training on how to use the system to get good coverage. So using static analysis simplified the setup considerably.

Plan vs. actual. The study did not go exactly as planned. The developer familiar with LbGrid was not available on the first and the last days of the study. Generally, the experimenter had limited access to the developer. We estimate the developer spent around 4 hours, including the initial meeting, designing and discussing the code architecture, answering occasional questions, examining snapshots and responding to our emails.

Target architecture. The experimenter met with the developer for two hours, and gave him an overview of the architectural views we were extracting. The developer said he used and liked tools that extracted class diagrams from code. The experimenter asked the developer to draw the designed runtime architecture for LbGrid. The experimenter wanted to use the designed

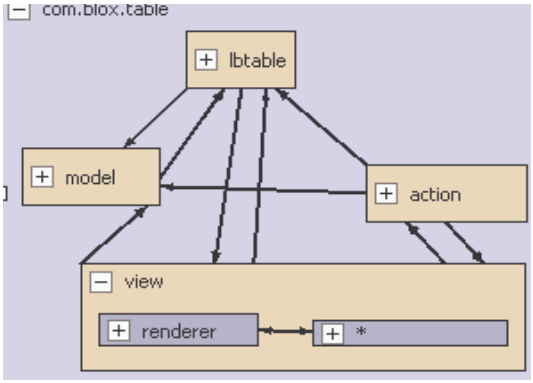


Figure 4.34: LbGrid: high-level module view, obtained using Lattix LDM (Lattix Inc 2008). A box represents a Java package.

architecture as a guide while adding the annotations, by following the same top-level architectural tiers and the same architectural decomposition. Unsurprisingly, the developer drew an abstracted class diagram showing the core types in LbGrid (Fig. 4.35).

4.8.4 Annotation and Extraction Process

We now discuss the process the experimenter followed to annotate LbGrid and extract an object graph.

Isolating the module. The experimenter configured several stop-analysis files to have the tools analyze only the compilation units from a list of selected packages and exclude others.

Annotation and extraction methodology. The experimenter used a tool to generate initial default ownership domain annotations for the selected Java files (See Appendix A.4.4). He then completed the annotations mostly manually, as we discuss in the next section. At times, he used a utility to globally find and replace annotations across several files. He then used mainly the two tools, ArchCheckJ (Section 4.3.1) and ArchRecJ (Section 4.3.2). He used ArchCheckJ to validate the annotations and ArchRecJ to extract OOGs.

Deciding on the annotations. The best annotations produce a view comparable to what an architect might draw for an architecture. Ideally, an architect familiar with the system would propose the runtime tiers for the system. In this case, it seemed that having the developer provide a target runtime architecture would be difficult, since he drew an abstracted code architecture. So, instead, the experimenter studied the developer's diagram, and suggested organizing the objects according to the following top-level domains: `UI`, `MODEL`, `LOGIC` and `DATA` (boxes with dashed borders in Fig. 4.36). The developer confirmed that the proposed architectural tiers seemed reasonable. Another senior developer who was familiar with other parts of the system also agreed with the high-level organization the experimenter proposed for the LbGrid architecture.

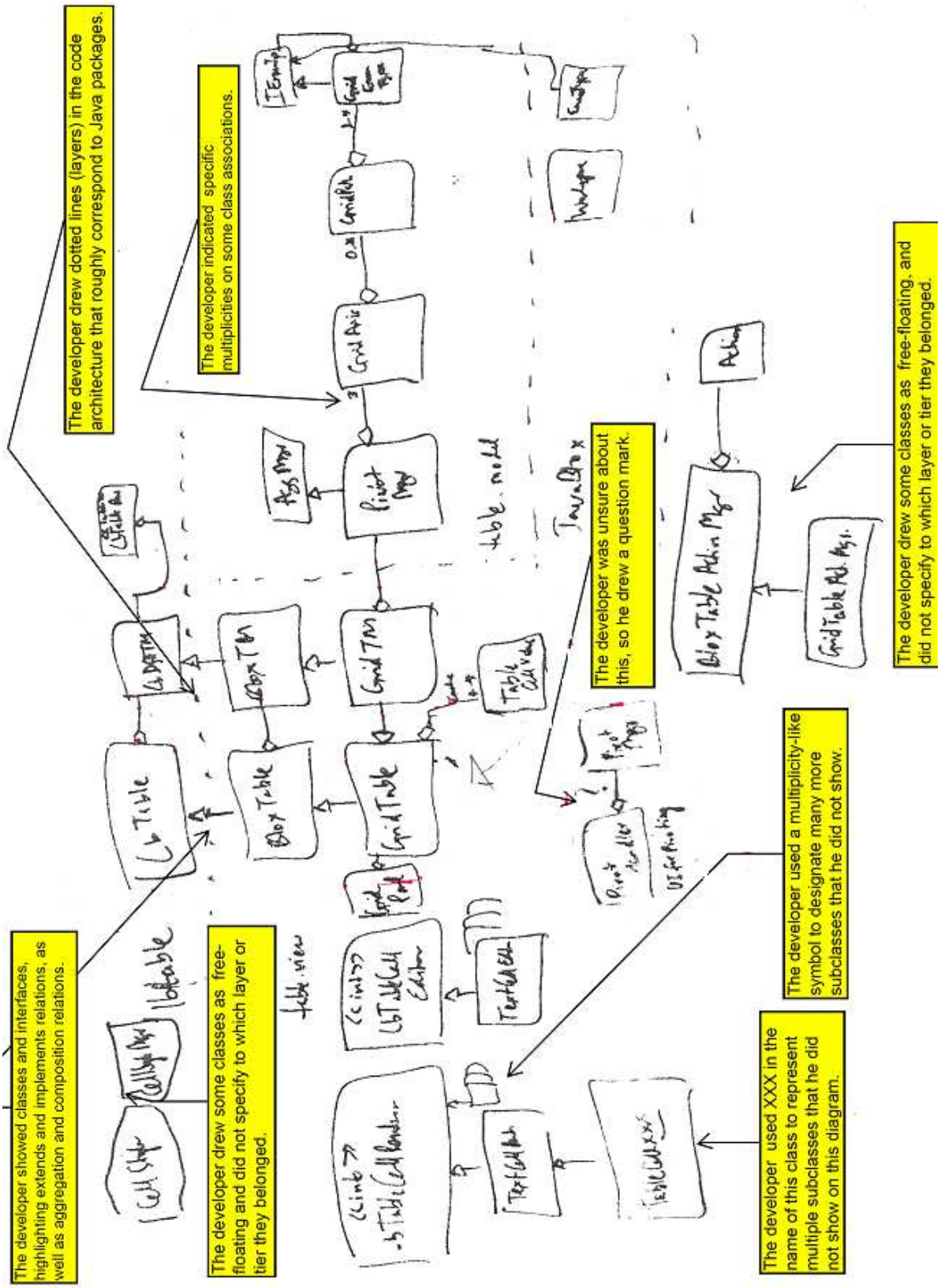


Figure 4.35: LbGrid: developer's diagram, which I annotated manually.

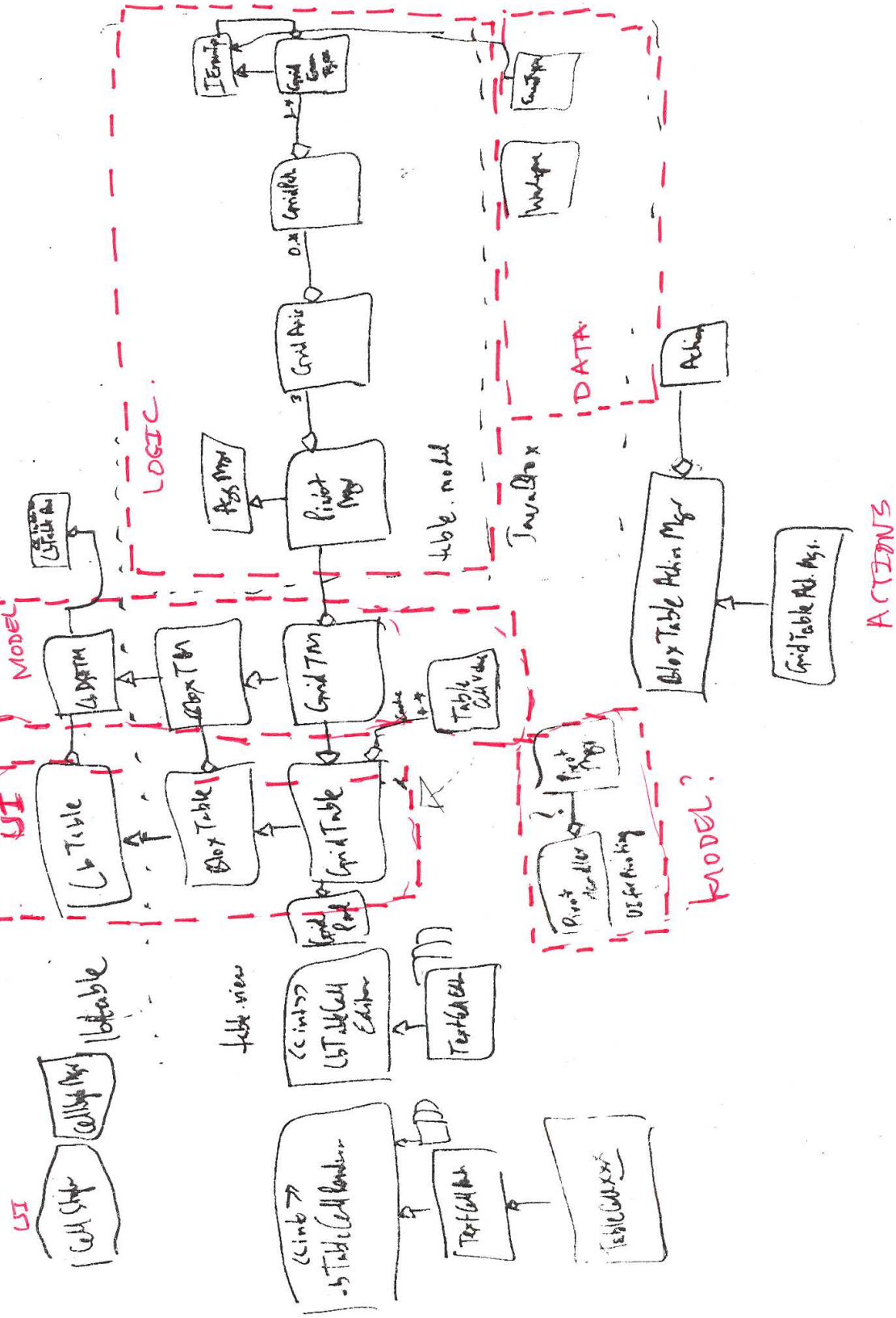


Figure 4.36: LbGrid: top-level domains which I suggested, shown with a dashed border.

Then, the experimenter started mapping objects to domains. As a first approximation, he mapped types to domains. Of course, not all the instances of a type, such as `List`, always appear in the same domain. Also, LbGrid has several classes that are instantiated only once, e.g., `Workspace`. In many cases, he used the package declaration as a guide. For instance, the experimenter often annotated an instance of a class declared in the `data` package to be in the `DATA` domain, or the corresponding `D` domain parameter. The trickier cases were instances of classes from nondescript utility packages that gave no indication about which runtime tier they belonged to. The experimenter organized the core types as follows:

- `UI:` instances of `LbTable`, etc.;
- `MODEL:` instances of `LbTableModel`, etc.;
- `LOGIC:` has instances of `PivotManager`, etc.;
- `DATA:` has instances of `Workspace`, `Predicate`, etc.

Once the experimenter figured out the top-level domains, he propagated them as domain parameters, as needed, using the mnemonic domain parameter names: `U` for `UI`, `M` for `MODEL`, `L` for `LOGIC`, and `D` for `DATA`.

Prioritizing the annotation warnings. The experimenter was not planning on adding domain links to LbGrid, so he turned off the corresponding checks for the duration of the field study. Otherwise, except for the implicit defaults or those added by the annotation defaulting tool, every reference type must be annotated. Enabling all the annotation checks at once would generate tens of thousands of warnings in the Eclipse problem window, and bring Eclipse to a standstill (the experimenter was running the tools on a modest Intel Pentium 4 (2 GHz) with 1.5 GB of memory). Moreover, one missing or incorrect annotation in the code could potentially produce several warnings. So the experimenter gradually enabled various annotation checks, and addressed annotation warnings from the most to the least important ones, as we discussed in Section 4.4.1 (Page 125)

Refining the annotations. In the early iterations, we placed most objects in one of the domain parameters, `U`, `M`, `L` or `D`. Since each domain parameter was transitively bound to a top-level domain, e.g., `U` to `UI`, `M` to `MODEL`, these early snapshots showed many objects in the top-level domains. But these early diagrams helped the experimenter refine the annotations and move a few objects between the top-level domains. In later iterations, he defined several private and public domains, and moved secondary several objects from a top-level domain to a private or public domain of a primary object, or passed objects linearly, to reduce the number of top-level objects, as we discussed in Section 4.4.2 (Page 126).

Strict encapsulation. The experimenter identified any encapsulated objects and placed them in private domains. As a first approximation, he recognized some of these objects if the containing class used them only inside its private representation, and did not have any accessors that returned them.

Logical containment. The developer's feedback helped the experimenter define several public domains with architecturally meaningful objects. Using logical containment often involved only

localized changes to the annotations. For instance, the public domain RENDERERS on LbTable holds objects of type TextCellRenderer and ColorCellRenderer. The EDITORS domain holds objects of type TextCellEditor and ColorCellEditor. In contrast, the module view shows all these types in one renderer package (Fig. 4.34).

The experimenter defined other architecturally significant public domains, such as:

- LbTableModel has a HEADERS domain to hold HeaderGridPath and HeaderGroup objects, among others;
- TableActionManager has an ACTIONS public domain for LockCellsAction and FillCellAction objects, etc.

Linear objects. He used the unique annotation where applicable. For example, LbGrid uses the following recurring pattern: a method performs a query, allocates a container to store the query result objects, then another object iterates the container elements then discards the container without storing a reference to it.

Questions to the developer. The experimenter had limited interaction with the developer. Occasionally, he asked the developer the following questions. The first question helped the experimenter identify objects that appear in the wrong conceptual tier. The second question guided the abstraction of the object graph by ownership hierarchy.

- Does this instance of type T belong to domain D?
- Within this domain D, is this object X conceptually part of this other object Y, so I can push X under Y?

The experimenter also asked the developer to identify the root class from which to derive the object graph. The developer pointed him to a unit test class.

4.8.5 Results

In this section, I discuss the field study results, in terms of the quantitative data we measured (Section 4.8.5.1), and the qualitative data we gathered during our interaction with the developer (Section 4.8.5.2).

4.8.5.1 Quantitative Data

Of the time spent on-site, the experimenter spent about 30 hours adding the annotations, type-checking them, and examining snapshots of the extracted object graphs. After the experimenter returned from the field trip, the developer emailed him some comments regarding one of the extracted object graphs. The experimenter spent another 5 hours adjusting the annotations to incorporate the developer's suggestions and address high-priority annotation warnings. At that point, the top-level object graph still did not fit on one letter-size readable page, such as the developer's code architecture (Fig. 4.37). There were still around 4,000 annotation warnings, most of them minor.

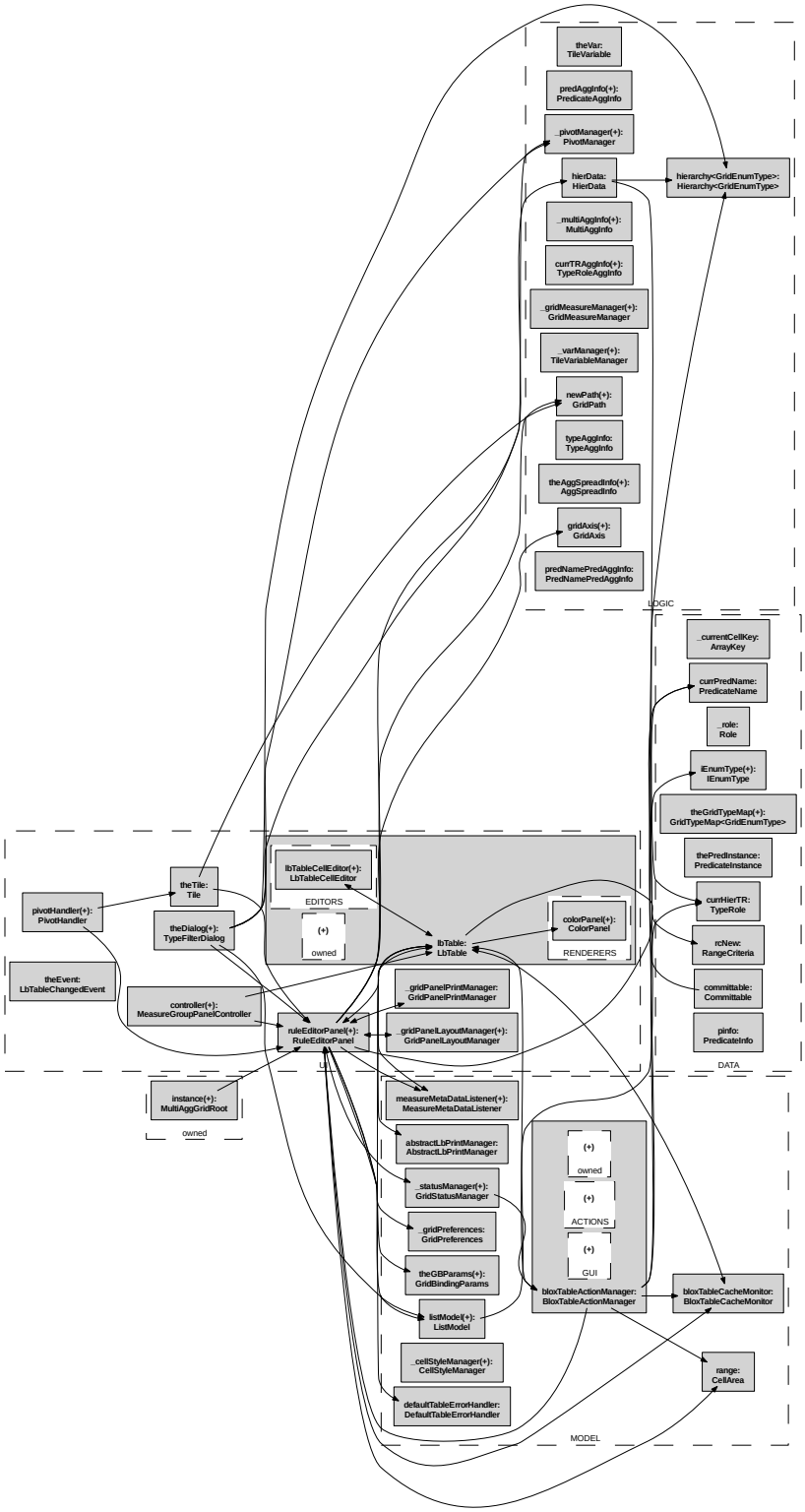


Figure 4.37: LbGrid: extracted object graph.

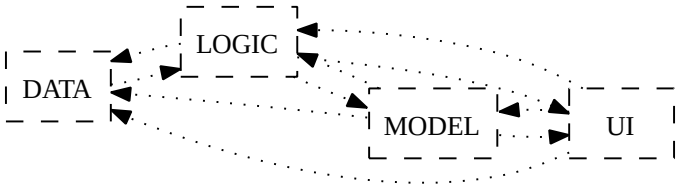


Figure 4.38: LbGrid: a 30-second high-level runtime view. The dotted edges summarize inter-tier references.

4.8.5.2 Qualitative Data

The field study allowed us to make the following qualitative observations.

Observation: The developer understood assigning objects to runtime tiers. The developer seemed comfortable thinking with a granularity coarser than an object or a class. He drew layers in his diagram that roughly correspond to packages, similarly to a high-level module view (Fig. 4.34). He understood mapping objects to domains, and even suggested moving some objects from one domain to another.

> "The following components should move to different containers: `AxisLayoutInfo` [from `MODEL`] to `LOGIC`."

Observation: The developer understood abstraction by ownership hierarchy. In particular, the goal is to show only architecturally significant objects in the top-level domains. The developer understood abstraction by ownership hierarchy, namely, pushing secondary objects underneath primary objects, as evidenced by his statement:

> "The following are too low-level to be at the outermost tier: `CellPosition`, ..."

For example, he recommended objects of type `TableHeaderGroup` and `TableHeaderGridPath` be pushed underneath the `LbTableModel` object in the `MODEL` tier. When provided with a printout of an extracted object graph, he expressed interest in viewing an object's sub-structure. At the time of the study, we did not have a standalone viewer. Since then, we implemented an interactive viewer that allows drilling into an object's substructure, zooming, scrolling and panning.

The developer also noticed when a top-level domain showed too many objects:

> "All components in `DATA` are also too low-level to be at the outermost tier, but I can't think of a larger component that you can expand to get to them. Not sure how to represent this."

To address the developer's last comment about the `DATA` tier, it is possible to elide a domain's structure, as in Fig. 4.38. The tool currently shows summary edges between collapsed domains. In future work, we will implement a feature to show edges between an object and a collapsed domain.

Observation: The developer understood object merging. By design, a SCHOLIA object graph conservatively merges into one object all the objects within a domain that may alias, based on their type information. For instance, a `BarChart` object in the `VIEW` domain merges the objects referenced through the `Listener` interface, the base class `AbstractChart` or its concrete subclass `BarChart`, because they may alias (Fig. 2.3(a)).

The ownership domain type system guarantees that two objects in different domains can never alias, however, so the analysis keeps those objects as separate.

Riehle posited that designers often use the following techniques to abstract their code architectures. They merge interface and abstract implementation class—although important for code reuse, such a code factoring is often unimportant from a design standpoint. They also subsume similar classes under representative classes, to avoid the clutter of showing many similar subclasses that vary in minor aspects (Riehle 2000, pp. 139–140). Indeed, the developer seems to have used the above techniques in his own class diagram. For example, he used "xxx" in the name of a few classes to represent multiple elided subclasses. He also used a multiplicity-like symbol to designate many more subclasses that he did not show on the diagram.

So it is unsurprising that these heuristics seemed also intuitive in a runtime view. However, SCHOLIA achieves similar results to those heuristics by merging objects in a domain based on their type, to soundly handle possible aliasing.

Observation: The extracted object graph shed some light into dark corners of the system. Upon examining an extracted object graph, the developer identified several classes that were candidates for deletion.

> "... `FormulaEditor` (will be deleted shortly)."

Observation: The developer seemed unsure about certain object communication. A developer often has a conceptual model of their architecture that is mostly accurate, but may be a simplification of reality (Murphy et al. 2001; Aldrich et al. 2002a). Indeed, the LbGrid developer drew some connections with question marks. An extracted object graph might help him confirm the presence or absence of communication.

Observation: A runtime view may help with certain coding tasks, but not with others. The developer was skeptical of the value of the extracted object graph (we recorded his opinion below before we gave him a standalone interactive viewer):

> "To step back a little and look at the diagram itself, so far, I can't see the value of a runtime view. I suspect that this will make more sense if I were to be able to drill down into the components. Or do you think that I should be able to see something in the outermost tier itself?"

We emphasized to the developer that the intent of a runtime view is to complement, not replace, a class diagram. Since he mentioned sequence diagrams, we explained to him that a sequence diagram is a kind of runtime view that shows method invocations for a specific use case, but it is not a complete runtime architecture. A more closely related diagram would be an object diagram which shows object instances exclusively, which (Gamma et al. 1994) use to

explain the standard design patterns. We suggested to the developer that he could think of an extracted object graph as a global object diagram, for the whole system, and one where each box is an aggregate of objects.

To address the developer's comment, we showed him how hierarchy enables obtaining a high-level object graph (Fig. 4.38), which makes explicit several global structural constraints that are implicit in the code, e.g., that objects in the `DATA` domain do not reference objects in the `MODEL` domain.

When reasoning about modifiability, a code architecture may be more helpful than a runtime architecture. The developer may have been focused on such tasks because he drew a detailed class diagram mostly from memory, and referred to Eclipse only occasionally to verify the name of a type. He seemed apologetic about the current design having many subclasses and a parallel inheritance hierarchy. In the current design, `GridTable` extends `BloxTable` extends `LbTable`. A parallel inheritance hierarchy exists between `GridTableManager`, `BloxTableManager` and `LbDefaultTableManager`.

He mentioned that one could refactor away some of those classes and move their functionality into their super-classes (the rationale for the current factoring is that super-classes are oblivious to accessing data from a workspace). He even asked the experimenter if he could think of a design that did not require proliferating sub-classes.

Since he was very familiar with the LbGrid code, he did not immediately see the value of a runtime architecture. We posit that because the runtime architecture abstracts away the factoring into interfaces, base classes and subclasses, it may actually be simpler to explain to a developer who is completely unfamiliar with the code, such as a new hire.

Observation: A runtime architecture can help explain listeners. A runtime architecture can help answer questions that a developer might have about an unfamiliar code base, such as: What instances point to what other instances? Thus, an object graph diagram can help explain what objects get notified during a change notification. In many cases, UML class diagrams or call graphs do not help answer such questions, because they do not show instances. For example, using the Listeners system (Chapter 2), an object diagram (Figs. 2.3(a), 2.3(b)) highlights the reference structure between `pieChart`, `barChart` and `model` better than a class diagram (Fig. 2.2).

LbGrid uses listeners heavily. Several classes have lists of listeners and implement various listener interfaces. Neither the developer's diagram nor an automatically generated class diagram, explain how the listeners work in LbGrid. We posit that this aspect of the architecture would be particularly challenging for a new hire. In future work, it may be useful to identify bug reports or enhancement requests that require understanding the listeners in LbGrid, and for which the extracted runtime architecture would be helpful.

Observation: Picking the right labels for architectural elements is crucial. Without careful labeling, developers may not recognize the models that a reverse engineering tool extracts (Murphy et al. 2001). Indeed, during the field study, the developer insisted on specific labels for the various tiers, e.g., use `UIMODEL` instead of `MODEL` (we still use `MODEL` here for consistency with prior documentation). In particular, with every OOG with which he was provided, he seemed to always visually scan the OOG, looking for instances of the core types from his class diagram:

"Where is `GridPanel`? I don't see it here."

Observation: The developer expected to see multiplicities on the object graph. Indeed, the developer's diagram has specific multiplicities on several associations. Many reverse engineering tools show multiplicities on class diagram. The developer suggested that showing this information on the object graph would be helpful, so this is a feature worth considering in future work. Of course, there are limits to the information that can be extracted statically from the code. In particular, a static object diagram is never going to show multiplicities that are as accurate as the ones in a dynamic object diagram, but the latter reflect only specific program runs.

Observation: The developer expected the tools to render a judgement on the recovered architecture. Many architectural extraction case studies evaluate the quality of a recovered architecture by computing some metrics, e.g., on dynamic coupling (Arisholm et al. 2004). Another avenue would be to check and measure the structural conformance of the built architecture against a designed one, but this requires establishing the target runtime architecture. This is the approach that SCHOLIA adopts, as described in the following chapters.

Observation: The developer seemed to favor an unsound abstracted task-specific view over a sound runtime architecture. A tool that extracts a class diagram automatically would show at least 300 classes for LbGrid, organized by packages. However, the developer's manual diagram had many fewer types. So the question is whether a runtime architecture should soundly reflect all objects and relations that exist at runtime, or only those that are of current interest to the developer. In a principled approach like SCHOLIA, the main abstraction technique is through the use of ownership hierarchy. A developer changes the annotations to push secondary objects under primary objects, and sometimes changes the code to support strict encapsulation. On the other hand, an unprincipled approach would allow a developer to elide any object or domain in the extracted architecture. In future work, we will consider ways to make a runtime architecture reflect more directly the types that are of interest to a developer, while maintaining soundness.

4.8.6 Validity

We identify the following confounding factors with the field study.

Experimenter bias. The experimenter understood ownership domain annotations and designed several of the tools that he used himself during the field study. Moreover, he had access to the code for the tools and customized them to the task to minimize data entry by loading settings from a file. However, a typechecker kept him honest, i.e., he could not just insert any annotation or manipulate the extracted object graphs. In a few instances, the experimenter backtracked on certain annotations he had just inserted.

Code unfamiliarity. The experimenter was completely unfamiliar with the code. A developer who is familiar with the code could perhaps add better annotations faster.

Developer motivation. The field study occurred in a workweek during which the developers were busy meeting a product ship deadline. As a result, they were less motivated to help the experimenter. Moreover, the developer seemed skeptical about the method and the tool.

Domain familiarity. LbGrid was somewhat similar to the JHotDraw subject system the experimenter studied previously, in that they are both GUI-based applications that used the Java Swing and AWT libraries. The experimenter also had some experience with the application domain, having previously developed a reusable grid control.

4.8.7 LbGrid Summary

The field study helped us confirm the following. First, we confirmed that an outside developer understood abstraction by ownership hierarchy and by types. Second, using only static analysis was compelling during architectural extraction, even without considering issues of soundness and the need to reflect all possible program runs. For a variety of reasons, it would have been difficult to setup and run the LbGrid system in order to use an architectural extraction method based on dynamic analysis. In addition, using a dynamic analysis would have required the experimenter to learn how to use the LbGrid system in order to get a good coverage. This would have been difficult because it would have required populating a database with appropriate test data, and learning how to navigate a fairly complex user interface with many user-selectable options. Finally, based on my own previous experience with ArchJava (Abi-Antoun and Coelho 2005; Abi-Antoun et al. 2007a), I could not have re-engineered LbGrid to ArchJava in the same few days that it took me to add the annotations, even after accounting for possible tool and language familiarity. Thus, adding annotations to an existing system seems more lightweight than re-engineering the system to use an extended language like ArchJava.

The goal of the field study was to better understand the process of adding the annotations. In addition, it would have been nice to demonstrate the value of the extracted architecture by showing how it can help identify undocumented information or contradict documented information, or help a developer in a typical code modification task. Due to the time constraints on the field study, we never got to concretely demonstrating the value of the extracted architecture.

4.9 Evaluation based on Cognitive Framework for Design

Table 4.1: Evaluation of the ArchRecJ tool based on the Cognitive Framework for Design (Storey et al. 1999).

Cognitive Design Element	**Corresponding feature in the tool**
Enhance bottom-up comprehension	
E1: Indicate syntactic and semantic relationships	Indicate logical containment or strict encapsulation; view field declarations that a DObject merges
E2: Reduce effects of delocalized plans	Handle inherited fields and domains; Show objects in actual domains bound to formal domain parameters
E3: Provide abstraction mechanism	Developer-specified annotations organize objects into groups (with merging) Hide all the private domains or the internals of a selected object
Enhance top-down comprehension	
E4: Support hypothesis driven comprehension	Start at selected root object and drill down; optionally visualize formals
E5: Provide overviews at various levels of abstraction	Limit depth of ownership tree and elide the “internals” of a selected DObject
Integrate bottom-up and top-down approaches	
E6: Provide views of multiple mental models	Show an approximation of the runtime structure at compile time
E7: Cross-reference multiple mental models	Label a DObject with a list of types; optionally show variable names
Facilitate navigation	
E8: Provide directional navigation	Navigate up and down ownership tree
E9: Provide arbitrary navigation	Search for a type, domain or field by name
Provide orientation cues	
E10: Indicate the current focus	Show currently selected element in ownership tree to the left of the visualization
E11: Display path that led to current focus	Show a nested graph starting from the root object
E12: Indicate options for further exploration	Show all domains, objects in domain; clicking on object selects it in ownership tree
Reduce disorientation effects	
E13: Reduce effort for user-interface adjustment	Main window shows the unfolding of the DGraph
E14: Provide effective presentation styles	The DGraph is laid out automatically; the tool supports filtering options

Table 4.1 presents an evaluation of the tool against a software visualization taxonomy used for software exploration tools, the Cognitive Framework for Design (Storey et al. 1999), with the usual disclaimers against self-evaluation.

Future work includes conducting additional evaluations (Nielsen and Mack 1994) in areas where visualizing the runtime architecture is crucial for program understanding, such as when tuning performance (Walker et al. 1998), or distributing an application (Spiegel 2002).

4.10 Discussion

We now discussion our evaluation of the annotations and the static analysis.

4.10.1 Research Questions (Revisited)

In this section, I discuss how well the evaluation answered the research questions (Section 4.2).

RQ1 – Precision: In practice, the static analysis does produce object graphs that have sufficient precision. The combination of precise generic types and domain annotations seems adequate in most cases.

RQ2 – Abstraction by ownership: In practice, a hierarchical object graph provides architectural abstraction by showing an order of magnitude fewer objects in the top-level domains, compared to a flat object graph.

RQ3 – Abstraction by types: In practice, abstraction by types achieve additional architectural abstraction in an object graph, even in the presence of a rich inheritance hierarchy, such as the one in JHotDraw.

RQ4 – Iteration: In practice, I was able to iterate effectively the process of adding the ownership annotations and extracting object graphs that have the desired architectural abstraction.

RQ5 – Annotations: In practice, I was able to add annotations that describe local, modular information. The process is iterative and self-correcting. I never encountered a situation where I was unsure of the annotation to add, needed the visualization to add the annotations, but the visualization itself needed the annotations. I could always add an annotation that typechecked, then go back and refine it as needed.

Moreover, the annotations that I added, e.g., `MODEL`, `VIEW`, `CONTROLLER`, were mostly natural and consistent with engineering intuition. In particular, I did not define fake domains such as `MODEL1` and `MODEL2` to compensate for the absence of an alias analysis or for the other sources of imprecision in the analysis (Section 2.6.3, Page 69).

RQ6 – Value: In practice, I indicated several instances of how an extracted OOG highlights potentially useful information about a system's runtime structure.

4.10.2 Evaluation Critique

Our evaluation of the object graph extraction suffers from a few weaknesses.

Subject system selection. One criticism is that we initially evaluated the approach on the same subject systems we used to develop the approach. For instance, the JHotDraw and the HillClimber case studies were formative. However, the LbGrid and Aphyds case studies were more summative.

Lack of comparison. Since there is other prior work in architectural recovery, a comparative validation would be useful. Ideally, one should apply several architectural recovery approaches to the same subject system (say, JHotDraw) and compare which one is less onerous, more direct, and qualitatively evaluate the output of the various tools. However, the only tool that claims to extract runtime architecture statically, X-RAY (Mendonça and Kramer 2001), supports only procedural code.

Missing target architecture. None of the subject systems we annotated came with an authoritative target architecture to guide the annotation process. Indeed, defining a reference or a target architecture is a research topic in its own right. In the case of LbGrid, I had access only to a rough guide based on a code architecture, so I was effectively defining the target runtime architecture during the process of adding the annotations.

Missing generic types. The code bases for JHotDraw and HillClimber did not already use generic types. In some cases, refactoring to generics was non-trivial, and uncovered some potential defects, e.g., when the same `Vector` object was used to store objects of different types, such as `String` and `Integer` objects.

Missing effort data. I conducted the formative JHotDraw and HillClimber case studies in different phases. In particular, I stopped the case studies in the early stages to fix several important bugs in the tool chain. So, I do not have accurate measures of the time spent adding annotations to those systems. However, during the field study, we did carefully measure the time needed to apply the approach, as we discussed above.

4.10.3 Soundness

All the subject systems we annotated still have several annotation warnings, which weakens the claims that the extracted object graphs are sound. Addressing these remaining annotation warnings could involve one of the following:

- **Increase type system expressiveness.** There are several known expressiveness challenges in the underlying type system. So one way to address those warnings is to extend the type system (See Section 9.2.2, Page 308);
- **Refactor the code.** In some cases, refactoring the code could allow for it to be annotated successfully, using the current type system. Of course, this is not an ideal solution. However, some of the code that cannot be annotated is also not following recommended practices of object-oriented design and programming. One class of warnings is due to the use of static fields, which are typically challenging for most ownership type systems.

Table 4.2: Performance measurements of the architectural extraction. LOC shows the lines of code. OOG measures the extraction time in minutes and seconds on an Intel Pentium 4 (3 GHz) with 2 GB of memory. WARN is the remaining annotation warnings. ABST indicates which abstraction by types was used.

System	LOC	OOG	ABST	WARN
JHotDraw	15,000	2:18	Trivial types / Design intent types	60
HillClimber	15,000	0:26	Trivial types	42

- **Inspect the code and suppress innocuous warning.** An annotation warning contributes to unsoundness in an extracted object graph only if eliminating the warning would result in new objects or edges in the object graph. One could manually inspect the problematic lines of code, and manually suppress the annotation if one can determine that such a warning does not make the object graph unsound. Still, one advantage of the approach in that case is that the developer does not have to inspect the entire code base.

4.10.4 Performance

Table 4.2 measures the execution time of the static analysis on several subject systems. The OOG time includes parsing the program's abstract syntax tree to retrieve the annotations and extracting the object graph. Overall, the OOG tool is sufficiently interactive to allow iteration.

4.10.5 Scalability

Since the biggest system we analyzed was only 30 KLOC (LbGrid), we cannot claim that we demonstrated SCHOLIA's scalability. However, when compared to many published architectural extraction case studies, even 30-KLOC does not fare too poorly.

4.11 Summary

I evaluated the object graph extraction analysis using several real medium-sized programs that I annotated manually. From an annotated program, I showed that I can use a tool to extract statically a hierarchical object graph that conveys meaningful abstractions. Indeed, these hierarchical object graphs seem to scale much more effectively than the corresponding flat object graphs that previous static analyses extract. In addition, an extracted object graph can give various insights by identifying undocumented information or contradicting manual documentation.

There are two questions, however, that the evaluation presented in this chapter does not answer. The first question is whether an extracted object graph corresponds truly to a standard runtime architecture. In Chapter 6, I discuss a separate analysis that raises the level of abstraction of an object graph, and abstracts it into a C&C view.

The second, perhaps more important question is when to stop iterating the process of refining the annotations and extracting OOGs. One strategy is to fine-tune the annotations in the code and the abstraction by types to make an extracted object graph similar to a posited architecture, to enable analyzing the conformance of the implementation to a designed architecture. In Chapter 6,

I make this evaluation criterion more precise by abstracting an extracted object graph into a built architecture, then analyzing the conformance of a built architecture to a designed one.

Analyzing conformance requires identifying and reconciling the key differences between the built and the target architectures, so the next chapter (Chapter 5) addresses the problem of synchronizing between two architectural views.

Credits

Wesley Coelho helped me re-engineer the HillClimber application to ArchJava (Abi-Antoun and Coelho 2005; Abi-Antoun et al. 2007a).

Acknowledgements

The author thanks Alan Mackworth for granting us permission to study the HillClimber code base and publish details of the case study. The author also thanks Molham Aref and the developers from LogicBlox Inc. for hosting the weeklong on-site LbGrid field study and granting us permission to publish details of the field study. My thesis committee offered especially timely and useful advice on how to conduct a field study. In addition, Mary Shaw, Thomas LaToza and Christopher Scaffidi gave us helpful comments on how to present the field study results.

Chapter 5

Architectural Synchronization[1]

5.1 Introduction

Software architects often face the problem of reconciling different versions of architectural models including differencing and sometimes merging architectural views—i.e., using the difference information from two versions to produce a new version that includes changes from both earlier versions. For instance, during analysis, a software architect may want to reconcile two Component-and-Connector (C&C) views representing two variants in a product line architecture (Chen et al. 2003). A runtime analysis could use the difference information to perform architectural repair (Dashofy et al. 2002). During evolution, the difference information between two versions can help focus regression testing efforts (Muccini et al. 2005).

Once the system is implemented, an architect may want to compare a designed C&C view against a C&C view retrieved from the implementation using various architectural extraction techniques. This is the approach that SCHOLIA takes to analyze communication integrity in the target architecture, following the *extract-abstract-check* strategy.

Several techniques have been proposed for differencing and merging architectural or design views. Most, however, do not detect differences based on structural information. Many assume that elements have unique identifiers (Alanen and Porres 2003; Ohst et al. 2003; Mehra et al. 2005). Others match two elements if both their labels and their types match (Chen et al. 2003), which is often infeasible when dealing with views at different levels of abstraction. Many techniques detect only a small number of differences. For instance, ArchDiff only detects insertions and deletions (van der Westhuizen and van der Hoek 2002; Chen et al. 2003), possibly leading to the loss of information when elements are renamed or moved across the hierarchy. Tracking changes, using element-level versioning, helps infer high-level operations, such as merges, splits or clones, in addition to the low-level operations, such as inserts and deletes (Jimenez 2005; Roshandel et al. 2004). But such an approach requires building new tools or changing existing tools to monitor the edits, and cannot handle legacy architectural models.

We propose an approach that overcomes some of these limitations, by differencing and merging architectural views based on structural information. In our approach, we leverage the hierarchy in the architectural views, and use a tree-to-tree correction algorithm to identify matches,

[1] Portions of this chapter appeared in (Abi-Antoun et al. 2006; Abi-Antoun et al. 2008)

and classify the changes between the two views. The algorithm uses the optional type information, whenever available, to avoid matching view elements that are incompatible, thus speeding execution and improving the match quality.

At the core of the approach is a polynomial-time tree-to-tree correction algorithm, MDIR, (Nahas 2009) that extends another optimal tree-to-tree correction algorithm for unordered labeled trees that detects renames, inserts and deletes (Torsello et al. 2005), and generalizes it to additionally detect restricted moves. The algorithm also supports forcing and preventing matches between elements in the views under comparison.

I developed a set of tools for the semi-automated synchronization of C&C views that uses the MDIR algorithm. The first tool, ArchJ2Acme, can synchronize a designed C&C view with a built C&C view retrieved from an ArchJava implementation. Another tool, ArchSynchro, can more generally synchronize two C&C views in Acme, regardless of how they were obtained. I evaluated the tools to find and reconcile interesting differences in real architectural views.

The chapter is organized as follows. Section 5.2 describes the challenges in differencing and merging architectural views, the underlying assumptions, and the limitations of our approach. Section 5.3 summarizes our novel tree-to-tree correction algorithm (Abi-Antoun et al. 2008). In Section 5.4, we use the algorithm to synchronize architectural C&C views. Section 5.5 illustrates the approach using extended examples on real architectural views.

5.2 Architectural View Differencing

A software architecture can generally be described as a graph, so differencing and merging architectural views 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 (Conte et al. 2004). 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 (Conte et al. 2004). Unique node labels enable processing graphs efficiently (Dickinson et al. 2004), which explains why many approaches make this assumption, e.g., (Alanen and Porres 2003; Ohst et al. 2003; Mehra et al. 2005). 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 (Messmer 1996; Conte et al. 2004). Non-optimal heuristic-based algorithms are more scalable, but often make restrictive assumptions. For instance, the Similarity Flooding Algorithm (SFA) "works for directed labeled graphs only. It degrades when labeling is uniform or undirected, or when nodes are less distinguishable. [It] does not perform well [...] on undirected graphs having no edge labels" (Melnik et al. 2002).

Several efficient algorithms have been proposed for trees, a strict hierarchical structure, so our

approach focuses on hierarchical architectural views. 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. Many approaches are hierarchical (Apiwattanapong et al. 2004; Raghavan et al. 2004; Xing and Stroulia 2005). So our choice is hardly new. However, we relax the constraints of existing approaches as follows:

No unique identifiers. Most techniques do not detect differences based on structural information. Many assume that elements have unique identifiers (Alanen and Porres 2003; Ohst et al. 2003; Mehra et al. 2005). Others match two elements if both their labels and their types match (Chen et al. 2003), which is often infeasible when dealing with views at different levels of abstraction. Making the assumption of having unique identifiers enables the use of exact and scalable algorithms that can handle thousands of nodes (Dickinson et al. 2004).

Unfortunately, architectural view elements often do not have unique identifiers. This is particularly the case for a built architecture extracted from an implementation using a tool. For maximum generality, SCHOLIA does not require elements to have unique identifiers.

No ordering. In the general case, an architectural view has no inherent ordering between 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, e.g., (Shasha and Zhang 1997). In comparison, tree-to-tree correction for unordered trees is MAX SNP-hard (Zhang and Jiang 1994). Some algorithms for unordered trees achieve polynomial-time complexity, either through heuristic methods, e.g., (Chawathe and Garcia-Molina 1997; Wang et al. 2003; Raghavan et al. 2004), or under additional assumptions, e.g., (Torsello et al. 2005).

Insertions and deletions only. Many architectural comparison techniques detect only a small number of differences. For instance, ArchDiff (van der Westhuizen and van der Hoek 2002; Chen et al. 2003) detects only insertions and deletions, possibly losing information when elements are renamed or moved across the hierarchy.

Name differences between two C&C views can arise for a variety of reasons. For instance, the architect may update a name in one view, and forget to update another view. Names are often modified during software development and maintenance. A name may turn out to be inappropriate or misleading due to either careless initial choice, or name conflicts from separately developed modules (Ammann and Cameron 1994). Furthermore, developers tend to avoid using names that may be in use by an implementation framework or library, a minor detail for the architect. Finally, architectural view elements may not have persistent names or their names may be generated automatically by tools.

This suggests that an algorithm should be able to match renamed elements. Identifying an element as being deleted then inserted when, in fact, it was renamed, would result in losing 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., (Spitznagel and Garlan 1998).

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 the hierarchy, by becoming children of the inserted nodes (Fig. 5.1).

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 adjust 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 a designed conceptual-level view with a 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 (Shaw and Garlan 1996).

No monitoring changes. Tracking changes, using element-level versioning, helps infer high-level operations, such as merges, splits or clones, in addition to the low-level operations, such as inserts and deletes (Jimenez 2005; Roshandel et al. 2004). But such an approach requires building new tools or changing existing tools to monitor the edits, and cannot handle legacy architectural models. For maximum generality, SCHOLIA assumes a disconnected and stateless operation.

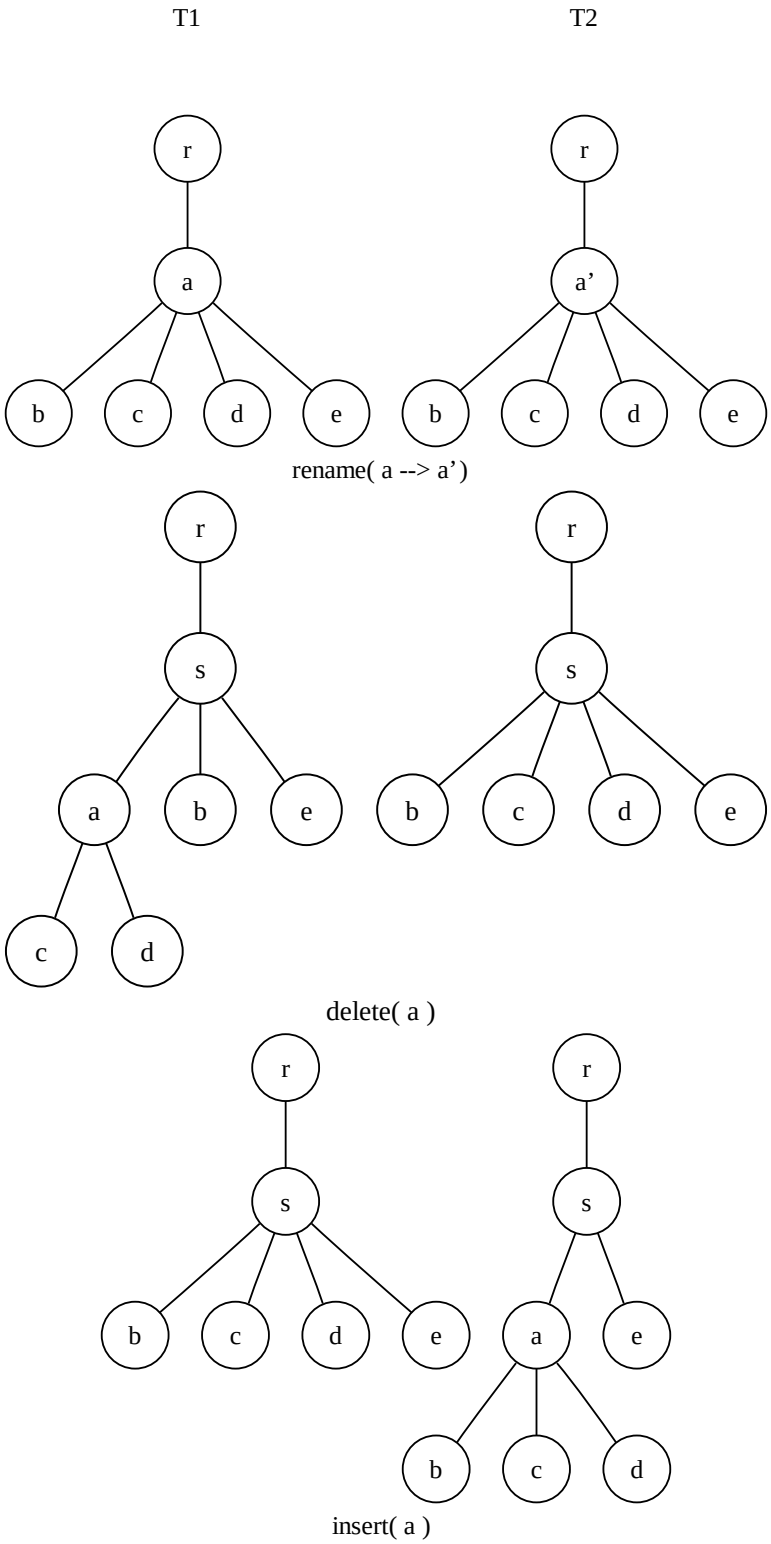


Figure 5.1: Tree edit operations.

Comparable views. 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. Checking the consistency of different but related views, such as a UML class diagram and a UML sequence diagram, is a problem in *view integration* (Egyed 2006), and is outside the scope of this dissertation.

No merging/splitting. Our approach does not currently detect the merging or splitting of view elements. Merging and splitting are common practice, but are difficult to formalize, since they affect connections in a context-dependent way (Erdogmus 1998). We leave merges and splits to future work.

5.3 Tree-to-Tree Correction

(Nahas 2009) developed an algorithm for the comparison of unordered labeled trees, MDIR (Moves-Deletes-Inserts-Renames), which generalizes a recent optimal tree-to-tree correction algorithm (Torsello et al. 2005), which we will refer to as THP. Here, we give an overview of the MDIR algorithm and leave the details, including its pseudo-code definition, elsewhere (Abi-Antoun et al. 2008; Nahas 2009).

5.3.1 Overview of Algorithm

We illustrate the MDIR algorithm on a small example of comparing two trees T_1 and T_2. MDIR exhaustively computes from bottom to top the cost of mapping each node in T_1 to every other node in T_2. The computed costs are stored in a cost matrix. Following the dynamic programming paradigm, MDIR uses the comparison on the high depth nodes to compare the low depth nodes. The example also illustrates the usefulness of the *successor set* approach, since bipartite matching cannot match subtree nodes, because of the need to preserve the hierarchical constraints.

MDIR starts by computing the cost of matching D to d (Fig. 5.2). Similarly, MDIR computes the costs of matching (D, e), (D, f), (D, g), ..., (E, d), (E, e), (E, g). Next, MDIR computes the cost of matching B to d (Fig. 5.3). Then, MDIR computes the cost of matching B to b (Fig. 5.4). This requires knowing the cost of the optimal *successor set mapping* for B and b. At this point, MDIR has computed the costs of matching every descendent of B to any node in the second tree, because of the post-ordering of the trees.

The optimal successor set mapping corresponding to the pair (B, b) is computed as follows (Fig. 5.5). First, take all the node pairs, where the first item is a descendent of B, and the second item is a descendent of b, i.e., the set $\{(D, d), (D, e), (E, d), (E, e)\}$. The optimal mapping will clearly be a subset of this set. To obtain that optimal mapping, we examine all mappings—except the ones that have been pruned because the bounds on their cost showed they could not be optimal. The other constraint is: if (x, y) is a pair in a mapping, neither x, nor y, nor any of their ascendents or descendents, can appear in any other pair in the same mapping. Thus, the

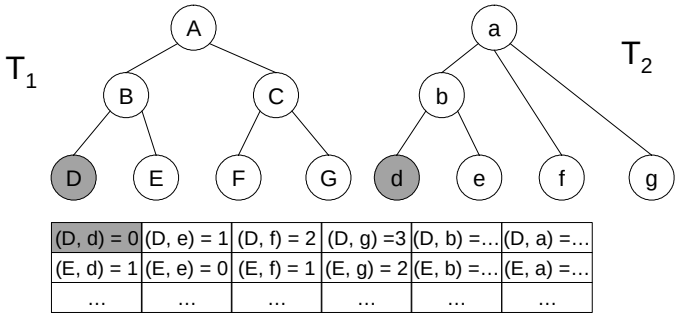


Figure 5.2: COST(D,d) = cost of editing label of D to d, i.e., the measure of similarity between the labels, in this case 0.

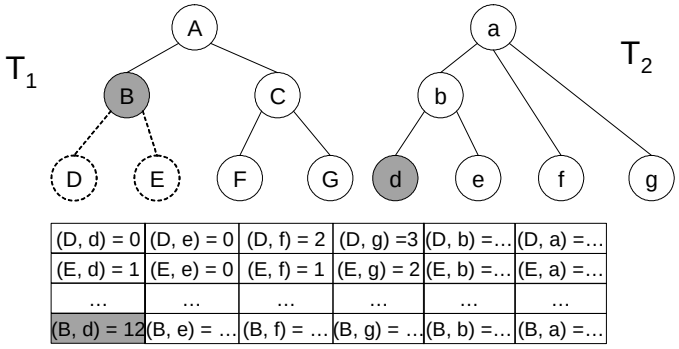


Figure 5.3: COST(B,d) = COST(deleting B's children) + COST(editing B's label). Assuming the cost of a deletion is 5 times a unit cost, COST(B, d) = COST(deleting D) + COST(deleting E) + COST(editing B's label) = 5 + 5 + 2.

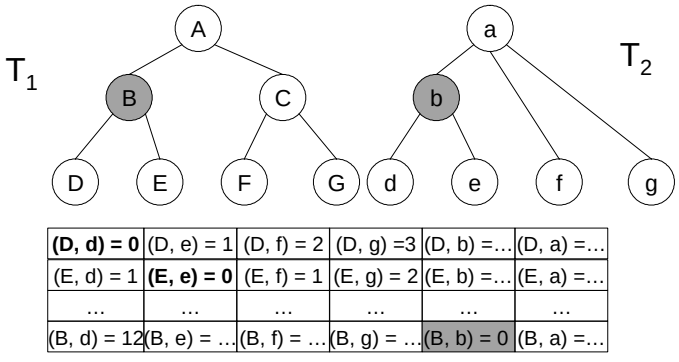


Figure 5.4: COST(B,b) = COST(successor set mapping of (B, b)) + COST(editing the label of B to b). COST(D,d) and COST(E,e) have been previously computed, thus COST(B, b) = COST(D,d) + COST(E,e) + 0.

optimal successor set mapping for (B, b) is $\{(D, d), (E, e)\}$. Finally, MDIR computes the cost of matching B to a (Fig. 5.6).

At the end of this phase, MDIR has determined the "best" successor set mapping, and stored it for the next phase, when MDIR will retrieve the best matches. MDIR could avoid keeping the optimal successor set mapping for each node pair in the first phase, to reduce the space

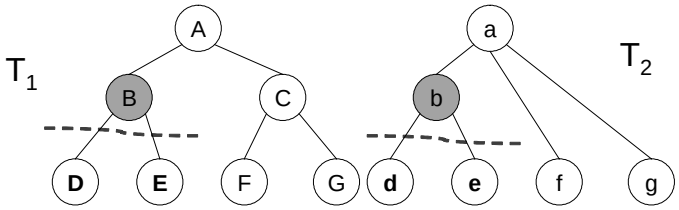
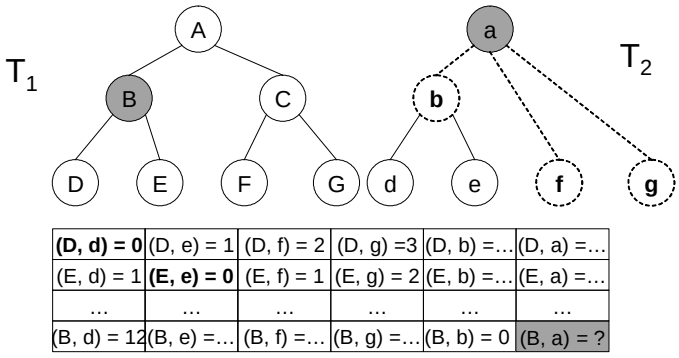


Figure 5.5: Computing the cost of matching B to b requires the *successor set mapping* of the pair (B, b). The *successor set mapping* of (B, b) is the set $\{(D, d), (E, e)\}$.



(D, d) = 0	(D, e) = 1	(D, f) = 2	(D, g) =3	(D, b) =…	(D, a) =…
(E, d) = 1	**(E, e) = 0**	(E, f) = 1	(E, g) = 2	(E, b) =…	(E, a) =…
…	…	…	…	…	…
(B, d) = 12	(B, e) =…	(B, f) =…	(B, g) =…	(B, b) = 0	(B, a) = ?

Figure 5.6: COST(B,a) = COST(successor set mapping of (B, a)) + COST(editing the label of B to a) + COST(deleting b, f and g).

complexity to $O(N^2)$. But it is simpler conceptually to store this information, and this is how we currently implemented MDIR.

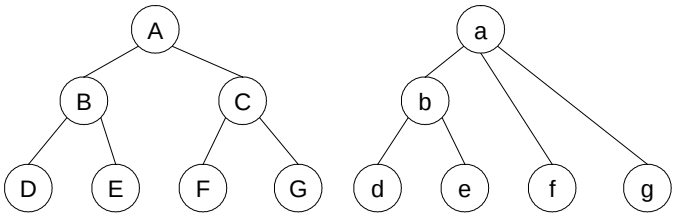
In the second phase, MDIR uses a recursive procedure to compute the match list, i.e., to determine what node corresponds to what other node. MDIR uses the following recursive formulation. The list of matches for subtree pair rooted at (x, y) consists of (x, y), in addition to the list of matches of each pair in the successor set mapping of (x, y).

MDIR starts with (A, a) (Fig. 5.7). The successor set mapping of (A, a) is $\{(B, b), (F, f), (G, g)\}$. So, MDIR first adds (A, a) to the match list, and then adds the pairs (B, b), (F, f), and (G, g) to the work list. Then, MDIR pops (B, b) from the work list, adds it to the match list, and adds to the work list the successor set (B, b), namely, (D, d) and (E, e). Next, MDIR pops (F, f) from the work list, adds it to the match list, and proceeds similarly.

5.3.2 Forcing and Preventing Matches

Manual overrides are not a standard operation in most tree-to-tree correction algorithms. MDIR has the ability to force and prevent matches between a node in tree T_1 and another node in tree T_2.

Preventing a match between two nodes i and j can be done by assigning a very large cost to the corresponding entry in the cost matrix $C[i][j]$. But forcing a match between two nodes is more difficult, due to the necessity of avoiding the deletion of the forcibly matched nodes and at the same time allowing the deletion of some of their ancestors. An explanation of the solution is in (Abi-Antoun et al. 2008; Nahas 2009).



Step	Work List	Match List
1	(A,a)	
2	(B,b)(F,f)(G,g)	(A,a)
3	(F,f)(G,g)(D,d)(E,e)	(A,a)(B,b)
4	(G,g)(D,d)(E,e)	(A,a)(B,b)(F,f)
5	(D,d)(E,e)	(A,a)(B,b)(F,f)(G,g)

Figure 5.7: Computing the match list.

5.3.3 Runtime and Memory Complexity

In practice[2], the observed runtime for MDIR is $O(KN^2)$, where K is a large constant. In comparison, THP has a worst case running time of $O(d^3N^2)$, where d is the maximum degree of a tree and $d << N$ (Torsello et al. 2005). Regarding memory requirements, both THP and MDIR could be implemented in $O(N^2)$ space, at the expense of additional complexity. Our current THP implementation requires $O(dN^2)$ space, and MDIR requires $O(bN^2)$ space, where b is a large constant factor.

5.4 Architectural View Synchronization

In this section, we discuss how we use a tree-to-tree correction algorithm to synchronize hierarchical graphs corresponding to C&C runtime architectures.

5.4.1 General Approach

We represent the structural information in a C&C view as a cross-linked tree structure that mirrors the hierarchical decomposition of a system. 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 the *defining occurrence*, and add a *cross-link* from each repeated node back to its defining occurrence. These redundant nodes, while they significantly increase

[2]A more formal analysis of the algorithm's complexity is in (Abi-Antoun et al. 2008; Nahas 2009).

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

Figure 5.8: Graphical overlays to indicate differences: Fig. 5.8(a) indicates a *match*; Fig. 5.8(b) indicates a *rename*; Fig. 5.8(c) indicates an *insertion*; and Fig. 5.8(d) indicates a *deletion*.

the tree sizes, greatly improve the tree-to-tree correction accuracy. 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 has the mapping between the nodes in the two graphs.

Synchronization is a five-step process: (1) setup the synchronization; (2) optionally view and match types; (3) view and match instances; (4) optionally view and modify the edit script; (5) confirm and optionally apply the edit script. The final step is optional because the architect may decline the edit operations for various reasons, or may be interested only in a change impact analysis (Krikhaar et al. 1999). Because Steps 1 and 5 are straightforward, we will only discuss Steps 2-4.

In Step 2, manually matching the type structures between the two views produces semantic information that speeds up the comparison. This optional information can also reduce the amount of data entry for assigning types to the elements that the edit script will create.

In Step 3, matching instances proceeds as follows: (a) build tree-structured data from the two C&C views to be compared; (b) use tree-to-tree correction to identify matches and structural differences; and (c) obtain an edit script to merge the two views.

The tool shows the structural differences by overlaying icons on the affected elements in each tree (Fig. 5.8). 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. The tool shows in italics ports that are inherited from the component type.

Various features can restrict the size of the trees and help reduce the comparison time:

- **Start at Component:** the user can select any component to be the tree root, and can reduce the tree sizes by selecting subtrees;
- **Restrict Tree Depth:** the user can exclude from comparison any nodes beyond a certain tree depth;
- **Elide Elements:** the user can exclude selected nodes and their entire subtrees from comparison. Elision is temporary and does not generate any edit actions.

The tool gives the user manual control using the following features:

- **Forced matches:** the user can manually force a match between two elements that may not match structurally;
- **Manual overrides:** the user can override any edit action suggested by the structural com-

parison.

Step 4 produces from the edit script a common supertree, that previews the merged view after the edit actions are applied. In this step, the user can assign types to elements to be created, change the types of existing elements, or override types that were automatically inferred based on the type matching in Step 2. The tool also checks the edit script for errors, such as illegal element names. The user can also rename any architectural element that the edit script will create. Finally, the user can cancel any unwanted edit actions.

5.4.2 Specialized Tools

This approach supports building tools for differencing and merging architectural views in a wide range of architecture description languages (ADLs). However, to evaluate our approach, we represent the C&C views in the Acme general purpose ADL (Garlan et al. 2000; Acme 2009).

We developed a tool to extract a built C&C architecture from an ArchJava implementation (Aldrich et al. 2002a). Similarly, one could extract built views from an implementation-constraining ADL with code generation capabilities, or an implementation-independent ADL with an implementation framework, such as C2 (Medvidovic and Taylor 2000).

We intended our synchronization tools to be lightweight enough that they can fit into a single dialog in an integrated development environment, such as Eclipse (Object Technology International, Inc. 2003), rather than require a specialized environment for architectural extraction (Telea et al. 2002). Both AcmeStudio, a domain-neutral architecture modeling environment for Acme (Schmerl and Garlan 2004; AcmeStudio 2009), and ArchJava's development environment, are Eclipse plugins, which reduced the tool integration effort.

We developed one tool, ArchJ2Acme, to make a designed architecture expressed in Acme, incrementally consistent with a built architecture extracted from an ArchJava implementation. In future work, the ArchJava infrastructure must change to support making incremental changes to an existing ArchJava implementation based on changes to the designed architecture.

We developed another tool, ArchSynchro, based on the same approach, to more generally synchronize any two C&C views represented in Acme. One view could correspond to a documented architecture. The second view could correspond to a C&C view recovered using any architectural extraction technique, e.g., (Schmerl et al. 2006). Alternatively, the second view could be another C&C view retrieved from a configuration management system, or one that corresponds to a variant in a product line.

Synchronizing a designed C&C view with a built C&C view must often address expressiveness gaps between architectural information at different levels of abstraction. Although we use Acme and ArchJava to illustrate some of these differences that must be bridged, synchronizing any pair of designed and built C&C views may encounter similar challenges.

Structural Differences. There will always be name differences of the same structural information between Acme and ArchJava. For instance, an ArchJava port can be named `in`, a reserved keyword in Acme. Even if code generation automatically produces a skeleton implementation from the architectural model, connector names and role names are lost, since ArchJava does not even name those elements. Finally, in Acme, port names are critical for typechecking. But in

ArchJava, port names are unimportant and obey the standard programming language notions of binding and scope.

Hierarchy. Acme treats hierarchy as design-time composition, where a component at one level in the hierarchy is just a transparent view of a more detailed decomposition specified by the representation of that component. Multiple representations for a given component or connector could correspond to alternative ways of decomposing an element. On the other hand, ArchJava views hierarchy in terms of integration of existing components, along with glue code, into a higher-level component. Due to the glue, a higher-level component is semantically more than the sum of its parts. These differing views of hierarchy create additional challenges for architectural synchronization. For example, if multiple representations are present at the design level, there must be a way to specify which of these representations was actually implemented.

Matching Instances. Obtaining the tree-structured data from Acme simply converts the Acme architectural graph into the cross-linked tree structure discussed earlier. Acme does not have first-class constructs for required and provided methods. In keeping with Acme's model for extensible properties, the tool adds properties on a port to represent its provided and required methods, as well as other salient properties, e.g., the port's visibility.

To obtain the tree-structured data from an ArchJava implementation, the ArchJ2Acme tool traverses the compilation units, ignores classes that are not component types, and fields that are not of component type. Different modeling choices are possible in this case. First, ArchJava does not name connectors or connector roles. The ArchJ2Acme tool generates synthetic names from the components and ports that a connector connects to. Second, the ArchJava top-level component can have ports, whereas the top-level component in Acme, i.e., the Acme system, cannot. One option is to create a top-level component in Acme to correspond to ArchJava's top-level component. Another is to create a synthetic component to hold these ports. Third, ArchJava ports can be private, whereas all Acme ports are public. One option is to represent ArchJava private ports as Acme ports on an internal component instance; another is to simply ignore private ports.

Matching Types. Assigning architectural styles and types to an Acme view enforces the architectural intent using constraints (Monroe 2001). For instance, a constraint on a component type may specify that all instances of that type must have exactly two ports. Similarly, setting architectural styles on the overall system—and on each sub-system representation if applicable, enforces any constraints associated with the style. In Acme, the Pipe-and-Filter style prohibits cycles, a constraint that a general purpose implementation language, such as ArchJava, does not directly enforce.

In many design languages, types are arbitrary logical predicates. An element is an instance of any type whose properties and rules it satisfies. And one type is a subtype of another, if the predicate of the first type implies the predicate of the second type. Such a type system is highly desirable at design time, because it allows designers to combine type specifications in flexible ways. Acme embodies this approach, but is hardly unique; for instance, PVS (Rushby et al. 1998) takes a similar approach. As an example of using a predicate-based type system, consider

an architecture that is a hybrid of the Pipe-and-Filter and Shared-Data architectural styles. In this example, a `Filter` component type has at least one `input` and one `output` port, while a `Client` component in the Shared-Data style has at least one port to communicate with the repository. A component in this architecture might inherit both the `Filter` and the `Client` specifications, yielding a component that has at least three ports—two for communicating with other filters and one for communicating with the `Repository`.

However, implementation-level type systems, such as the ones provided by C2SADL (Medvidovic et al. 1996) or ArchJava, cannot express the example above. A specification that a component has a port implies a requirement that the environment will match that port up with some other component. Therefore, conventional type systems require a component type to list all of the ports it might possibly have—or at least all those ports that are expected to be connected at runtime. There is no way to express that a `Filter` component has "at least two ports"—instead, one must say that the `Filter` has "at most" or "exactly" two ports. Therefore, in the implementation, one cannot combine the `Filter` type with a `Repository` component type—which defines a third port that is prohibited by the filter specification.

So a design-level predicate-based type system is fundamentally incompatible with a type system for a programming language. As a result, the matching algorithm may not rely on exactly matching typing information as in UMLDiff (Xing and Stroulia 2005). In our approach, the user specifies arbitrary matches between the type hierarchies from Acme and ArchJava, flattened and shown side-by-side.

Consider synchronizing the Acme model of a simple system following the Pipe-and-Filter style with its ArchJava implementation. In Fig. 5.9, the user matches the types as follows. The user selects the `Capitalize`, `CharBuffer`, `Lower`, `Merge`, `Split`, `Upper` component types in ArchJava and matches them with `Filter` Acme type. All the component instances of these ArchJava types will be assigned the `Filter` Acme type during synchronization. Using a limited form of wildcards, the user assigns the Acme type `Pipe` to the ArchJava connector type `ANY`. So any Acme connector created for an implicit ArchJava connector instance will have that type.

Since ArchJava ports are not typed, the user can individually assign to an ArchJava port a set of Acme port types. To reduce the manual work, the user uses another form of wildcards. He can assign an Acme type, e.g., `outputT`, to any ArchJava port that only provides methods. Similarly, he can assign the `inputT` Acme type to any ArchJava port that only requires methods. In addition, AcmeStudio defines connection patterns for most architectural styles. Based on these patterns, ArchJ2Acme can infer the Acme role types, once the user assigns types to components, ports and connectors. For instance, the tool infers the role type `sourceT`, based on the source component type `Filter`, source port type `inputT`, and connector type `Pipe`.

In this case, the synchronization produces the edit script in Fig. 5.10. Since the user mapped the types, the edit script elements already have types. Each view element that already has a type is displayed using the same type- and style- dependent visualization that it would have in AcmeStudio. If the user does not specify architectural types and styles, the elements that the edit script will create will be untyped. Of course, the user can set the types on the newly inserted view elements at a later point in AcmeStudio. Although assigning types during synchronization seems to duplicate functionality, it may affect the edit script and the view merging as explained below.

For instance, when a component instance is assigned the `Filter` component type, it inherits

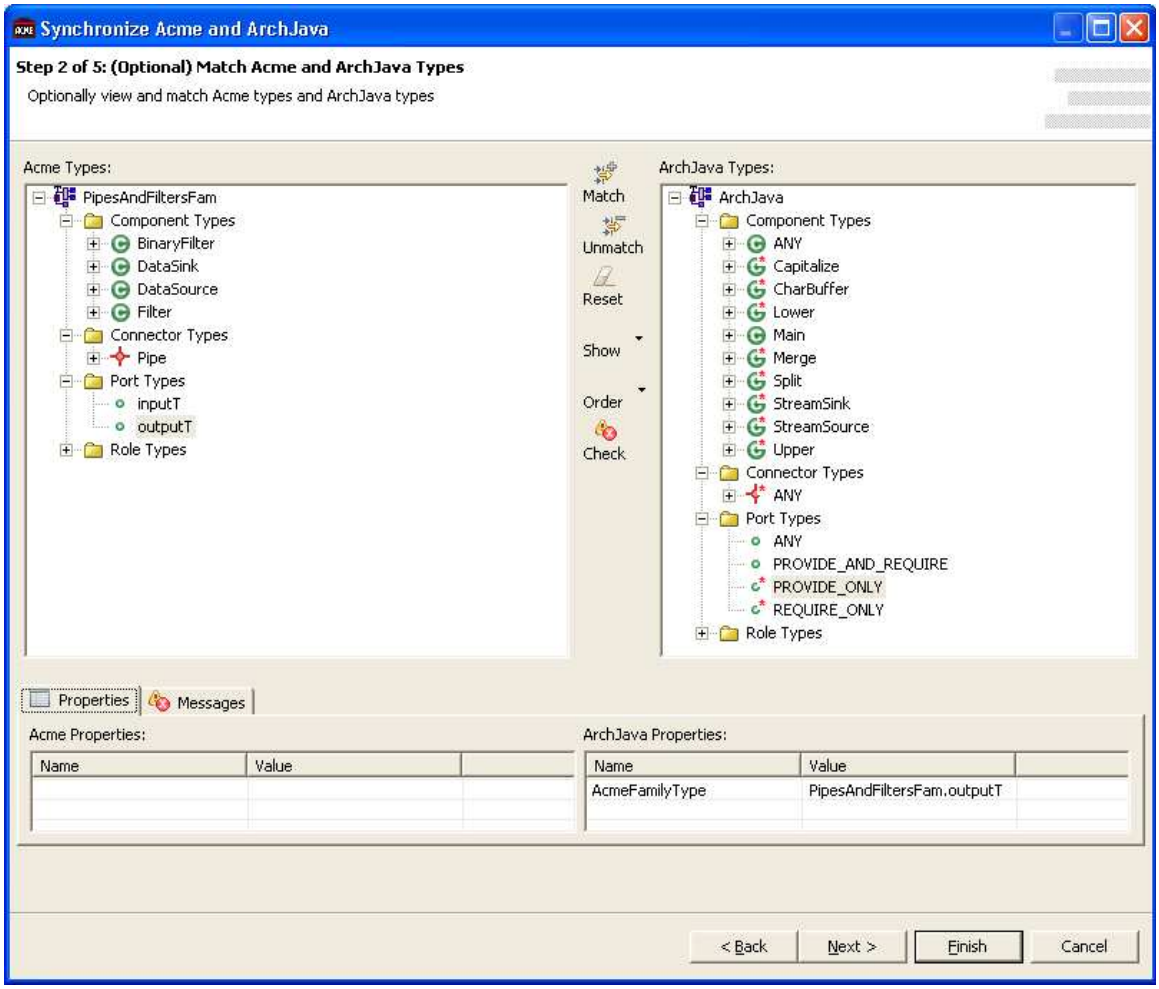


Figure 5.9: Matching types between the designed Acme model of a simple system following the Pipe-and-Filter style with its built ArchJava implementation.

any ports declared on that type, e.g., ports `input` and `output`, of types `inputT` and `outputT`. So ArchJ2Acme need not create additional ports of these types on the component instance. Based on the user's selection in Fig. 5.9, the tool matches the ArchJava port `portOut`—since it only provides methods, with the Acme type `outputT`. The tool suggests renaming the port `portOut` of type `outputT`, to match the `output` port on the `Filter` type.

The user can accept the corrective actions suggested by the tool using the Auto-Correct button in Fig. 5.10. In that case, the tool automatically renames `portOut` port to `output`, and updates all the cross-references in the edit script. The user can also change the assigned or inferred types before pushing the changes to the Acme model.

5.5 Evaluation

In this section, we evaluate the tools for C&C view synchronization in several extended examples on real architectural views. Our evaluation aims to answer the following hypothesis from

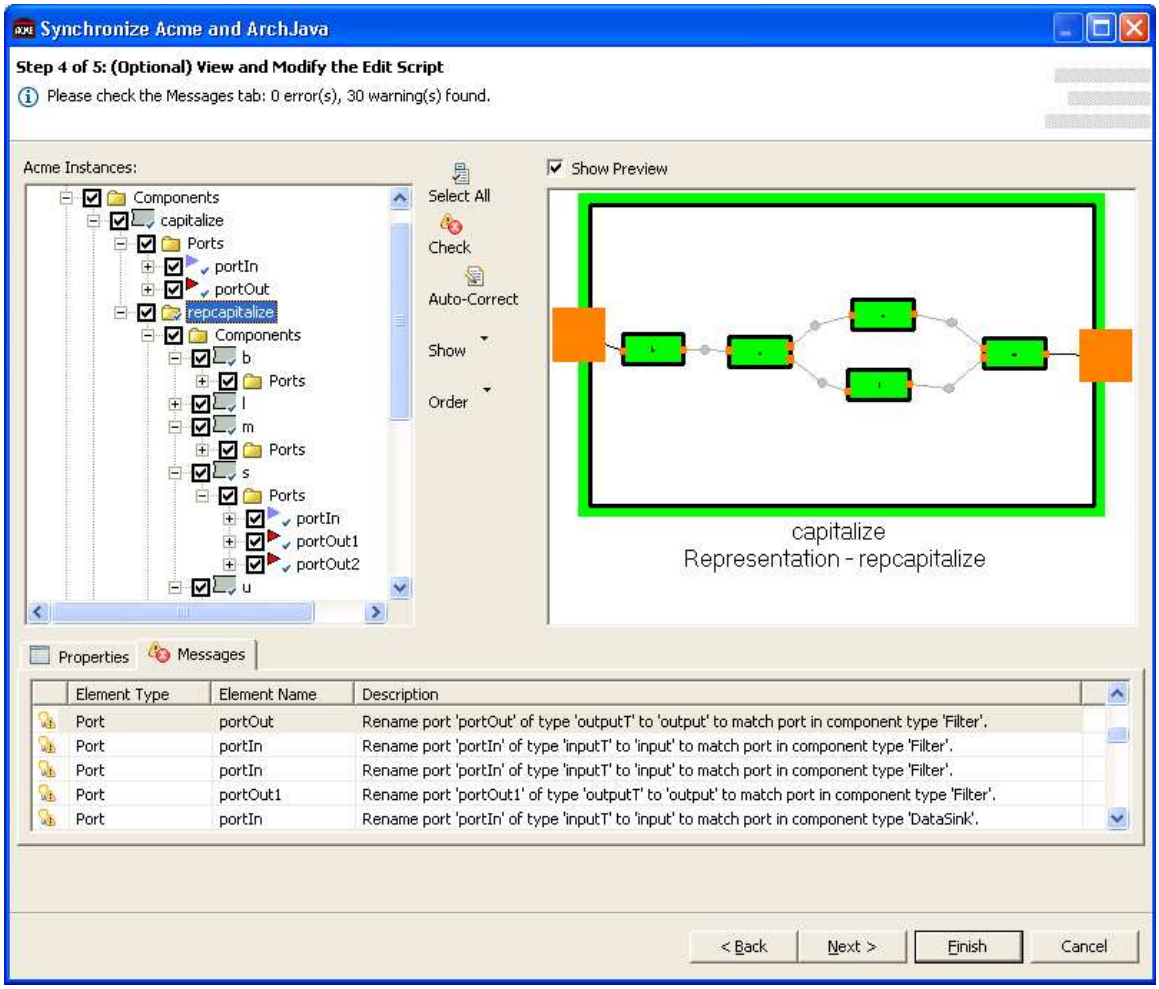


Figure 5.10: Validating the edit script can involve renaming some ports to match the names declared in the Acme type.

Section 1.10.

> *H-5: An analysis can structurally compare the built architecture to a documented target runtime architecture.*

We refine the hypothesis into the following research questions:

RQ2 – Comparison: *Can the structural comparison meaningfully compare a built architecture extracted from the implementation to a designed architecture?* The measurable criteria here are to minimize the occurrences where a developer must manually force or prevent matches between the view elements.

We now present three extended examples: AphydsAJ (Section 5.5.1), Duke's Bank (Section 5.5.2) and HillClimberAJ (Section 5.5.3).

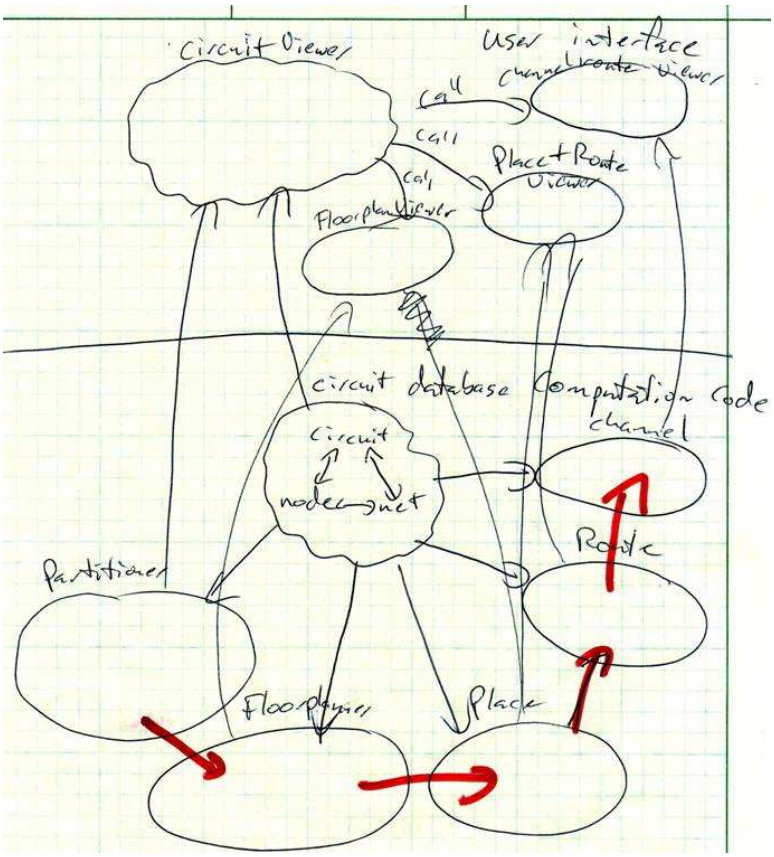


Figure 5.11: Aphyds: informal designed architecture drawn by the original developer. Source: (Aldrich et al. 2002a).

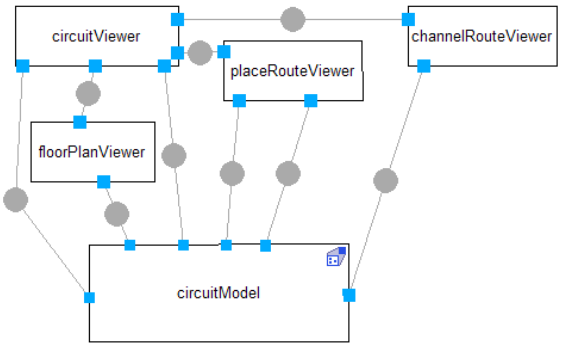
5.5.1 Extended Example: AphydsAJ

In this example, we synchronize a designed C&C view with a built C&C view retrieved from an implementation. This example mainly highlights the ability of the underlying MDIR algorithm to detect inserts, deletes and renames.

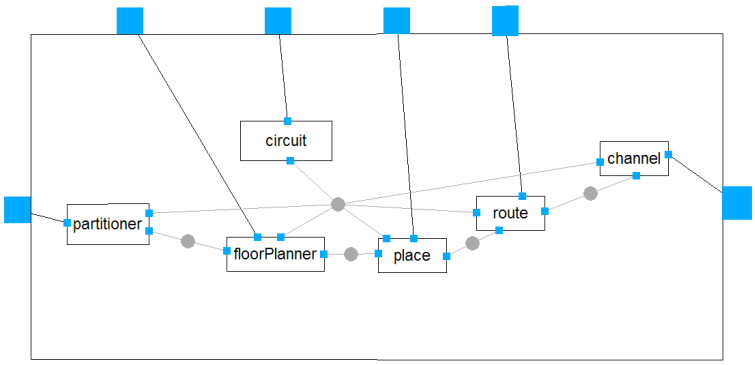
In Chapter 1, I introduced the Aphyds system. (Aldrich et al. 2002a) re-engineered the original Aphyds Java implementation into an ArchJava implementation to evaluate ArchJava's expressiveness to specify the architecture in code. In this chapter, we refer to that version as AphydsAJ. We use AphydsAJ since it has a documented designed architecture, and we can use the ArchJ2Acme tool to extract a built C&C view from the ArchJava implementation.

In the following discussion, I refer to the person who conducted the evaluation, i.e., myself, as the *experimenter*. The *developer* is the person who developed the code being analyzed. The experimenter has no prior experience with the original Java program, or with the process of re-engineering the Java program into the ArchJava implementation.

Designed Architecture. The developer of the original Java program informally drew the designed architecture (Fig. 5.11). The experimenter created an Acme model based on the informal architecture (Fig. 5.12(a)). He represented the `circuitModel` as a single component, and added



(a) Top-level Acme model.



(b) Acme representation of the `circuitModel` component.

Figure 5.12: AphydsAJ: designed architecture represented in Acme.

all the computational components to a representation of `circuitModel` (Fig. 5.12(b)). In the original diagram (Fig. 5.11), the thin arrows represent control flow, and the thick arrows represent data flow, but the experimenter did not make that distinction in Fig. 5.12 and showed all communication as Acme connectors.

Matching Types. The experimenter chose an Acme Model-View-Controller style, `MVCFam`. Since he was interested in the control flow, he assigned the `provideT` Acme port type defined in `MVCFam` to any ArchJava port that only provides methods. Similarly, he assigned the `useT` Acme port type to any ArchJava port that only requires methods, and the `provreqT` Acme port type to any ArchJava port that both provides and requires methods. He also assigned the generic `TierNodeT` Acme type to all components and the `CallReturnT` Acme type to all the implicit ArchJava connectors (See Fig. 5.13).

Matching Instances. The experimenter used the ArchJ2Acme tool to compare the two views. As he was the least sure about how he represented the `circuitModel` component in Acme, he decided to focus on that component first.

The ArchJ2Acme tool detected a few renames, e.g., ArchJava uses `model` instead of `circuitModel`, and inside that representation, ArchJava uses `globalRouter` instead of `route`

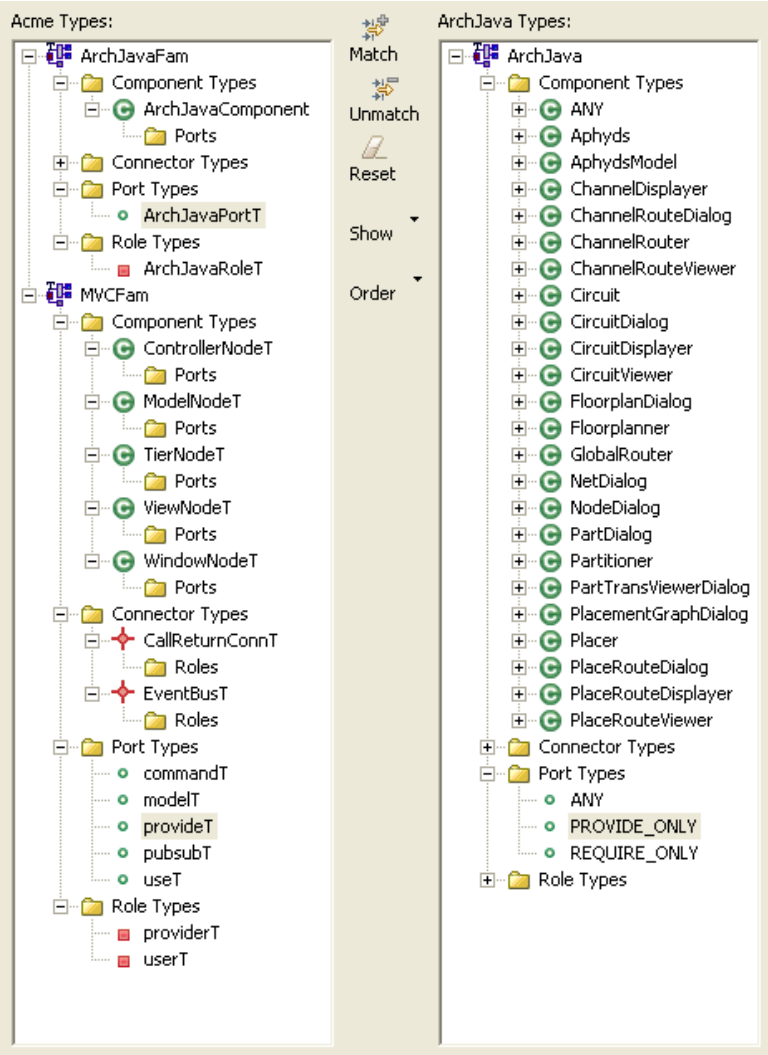


Figure 5.13: AphydsAJ: matching types between Acme (left) and ArchJava (right).

(Fig. 5.14). The experimenter was particularly intrigued that the Acme representation for `circuitModel` had more connectors than the ArchJava implementation. In Fig. 5.14, the tool only matched the `starConnector` which connects components `circuit`, `partitioner`, `floorPlanner`, `place`, `route` and `channel` (Fig. 5.12). The experimenter investigated this further and confirmed that the Acme connectors corresponding to the thick data flow arrows in the informal diagram (Fig. 5.11) are not in the implementation. Since Aphyds was written for academic study and not for industrial application, it is missing some of the data flows that would be present in a real application, i.e., the data flow is simulated rather than real. So the experimenter accepted the edit actions to delete these extra connectors from the Acme model.

Merging Instances. The experimenter next turned his attention to the additional top level component, shown as `privateAphyds` (Fig. 5.14). Based on the synchronization options he selected, he determined that the tool created `privateAphyds` to represent a private `window` port in Arch-

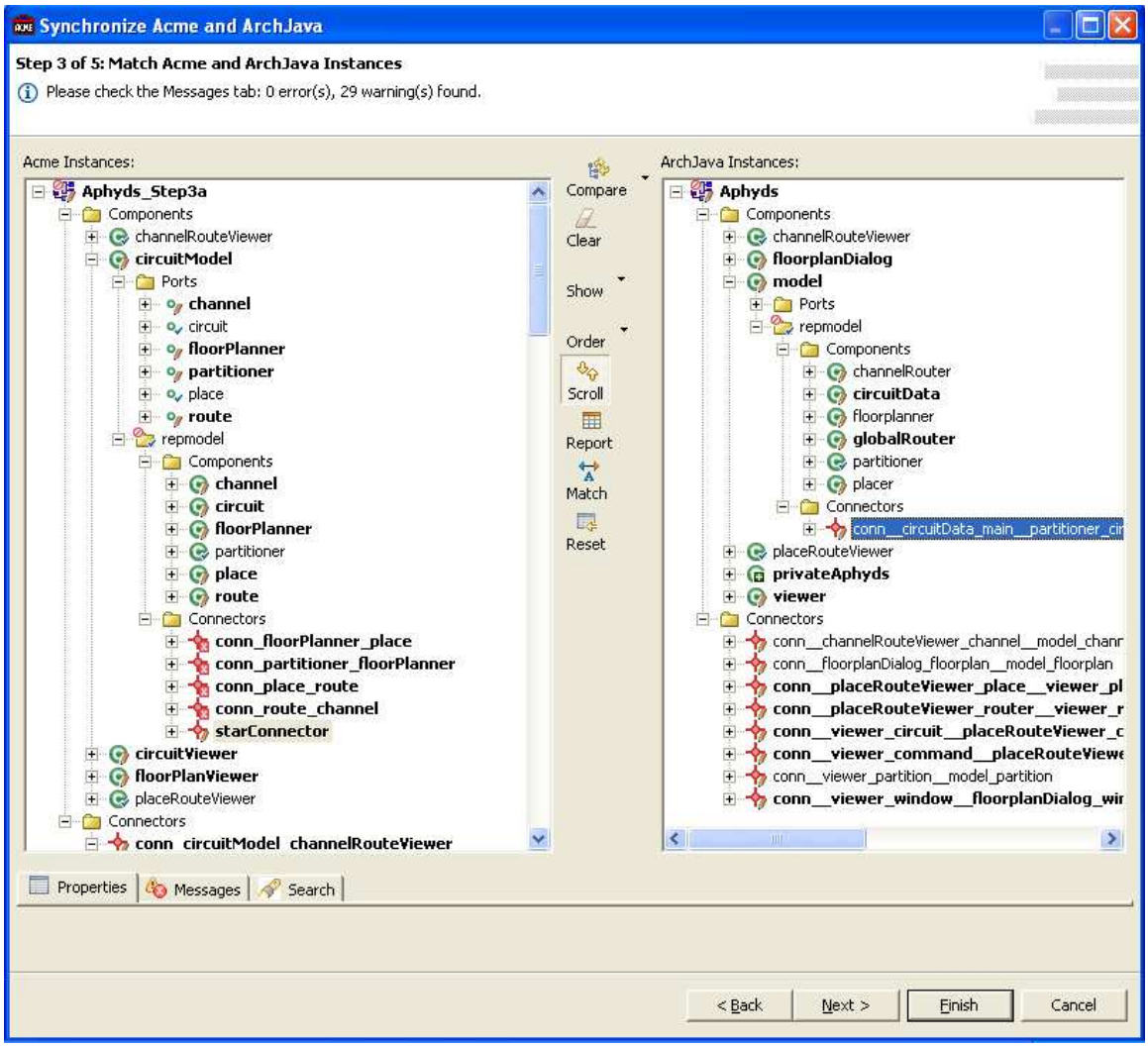


Figure 5.14: AphydsAJ: comparison of Acme C&C view (left) and ArchJava C&C view (right). The connector `starConnector` matches a connector in ArchJava with an automatically generated name (highlighted nodes). The component `privateAphyds` exists in ArchJava but not in Acme.

Java and the corresponding glue. After looking at the control flow, the experimenter assigned that subsystem the Publish-Subscribe Acme style. He also renamed component `privateAphyds` to `window`, renamed the added connector to `windowBus`, and assigned `windowBus` the `EventBusT` connector type from the style. The experimenter also decided to use the same component names as the ArchJava implementation to avoid future confusion, so he accepted the renames in the edit script.

Discussion. The experimenter manually laid out the resulting C&C view in AcmeStudio (Fig. 5.15). Unlike the original architect's view, Fig. 5.15 shows bi-directional communication taking place between components `placeRouteViewer` and `model`. The experimenter investigated that unexpected communication, and traced it to a callback. Aphyds is a multi-threaded

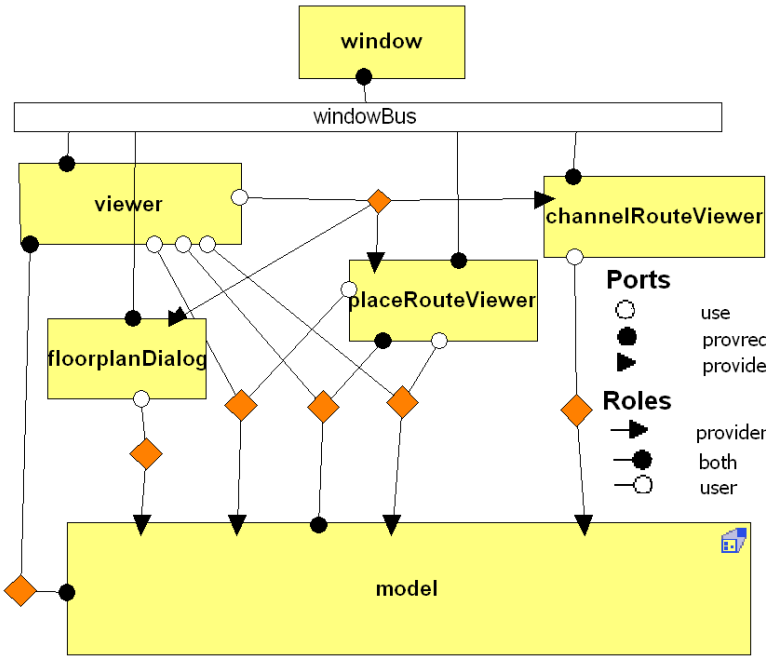


Figure 5.15: AphydsAJ: built architecture with Acme styles and types.

application with long running operations moved onto worker threads. So the experimenter noted that developers should not carelessly add callbacks from a worker thread onto the user interface thread.

Performance Evaluation. On an Intel Pentium 4 CPU 3GHz with 1.5GB of RAM, comparing an Acme tree of around 650 nodes with an ArchJava tree of around 1,150 nodes (Fig. 5.14) with MDIR took under 2 minutes. In comparison, THP took around 30 seconds, but produced less accurate results. In particular, THP did not treat component `privateAphyds` as an insertion, and mismatched all the top-level components. For AphydsAJ, the edit script consisted of over 300 renames, over 600 inserts and over 100 deletes.

5.5.2 Extended Example: Duke's Bank

In this example, we synchronize two C&C views, where the built view is recovered by instrumenting the running system. This example mainly highlights the ability of the underlying MDIR algorithm to detect moves in addition to renames.

The subject system is Duke's Bank, a simple Enterprise JavaBeans (EJB) banking application. The experimenter wanted to compare the documented architecture with the built architecture, recovered using an architectural extraction technique other than ArchJava. Duke's Bank is also representative of industrial code that uses middleware, and furthermore, has a documented designed architecture.

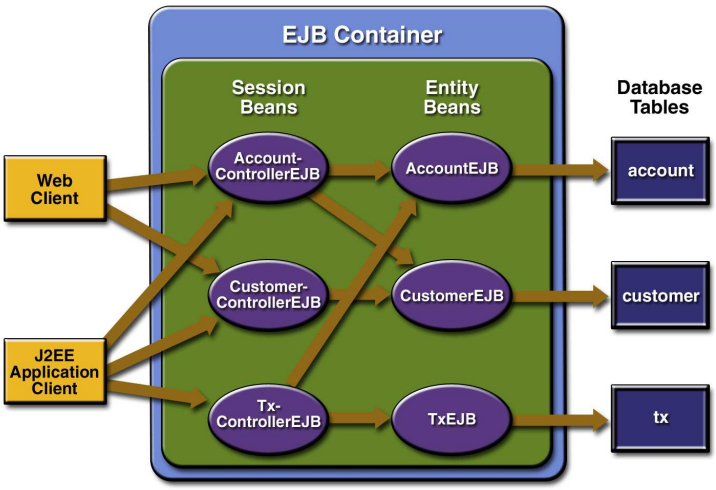


Figure 5.16: Duke's Bank: informal designed architecture (Sun Microsystems 2006).

Designed Architecture. The experimenter converted an informal diagram (Fig. 5.16) into an Acme model (Fig. 5.17).

Built Architecture. The built architecture was recovered by a dynamic architecture extraction tool, DISCOTECT (Schmerl et al. 2006). DISCOTECT currently generates one component instance for each session and entity bean instance created at runtime. So the experimenter post-processed the dynamically recovered architecture, and unified such multiple instances into one instance. The goal was to make the recovered C&C view in Fig. 5.18 comparable to a typical C&C view, where each component instance represents any number of runtime components.

Matching Types. In this case, the built view and the designed views use the same architectural style and types, so the experimenter skipped the optional step of matching types.

Matching Instances. The ArchSynchro tool correctly detected the moves corresponding to replacing the `container` component in one view with its representation in the other view (Fig. 5.19). Because a tool generated the names in the recovered view, e.g., `AccountBean_e55d75`, there was a large number of renames in this case. The ArchSynchro tool matched all the elements between the two views, despite the large number of renames.

Discussion. ArchSynchro also identified on `Account_Controller_Bean` a port that was attached to a `DbWriter` connector. Fig. 5.17 does not show a connection between the `Account_Controller_Bean` and the `DB` components. In fact, the EJB specification recommends that all database access goes through entity beans. In this case, the tool found an architectural violation in Sun's own example!

Performance Evaluation. On an Intel Pentium 4 CPU 3GHz with 1.5GB of RAM, MDIR took around 30 seconds to compare the two Acme trees, one with around 330 nodes, and one

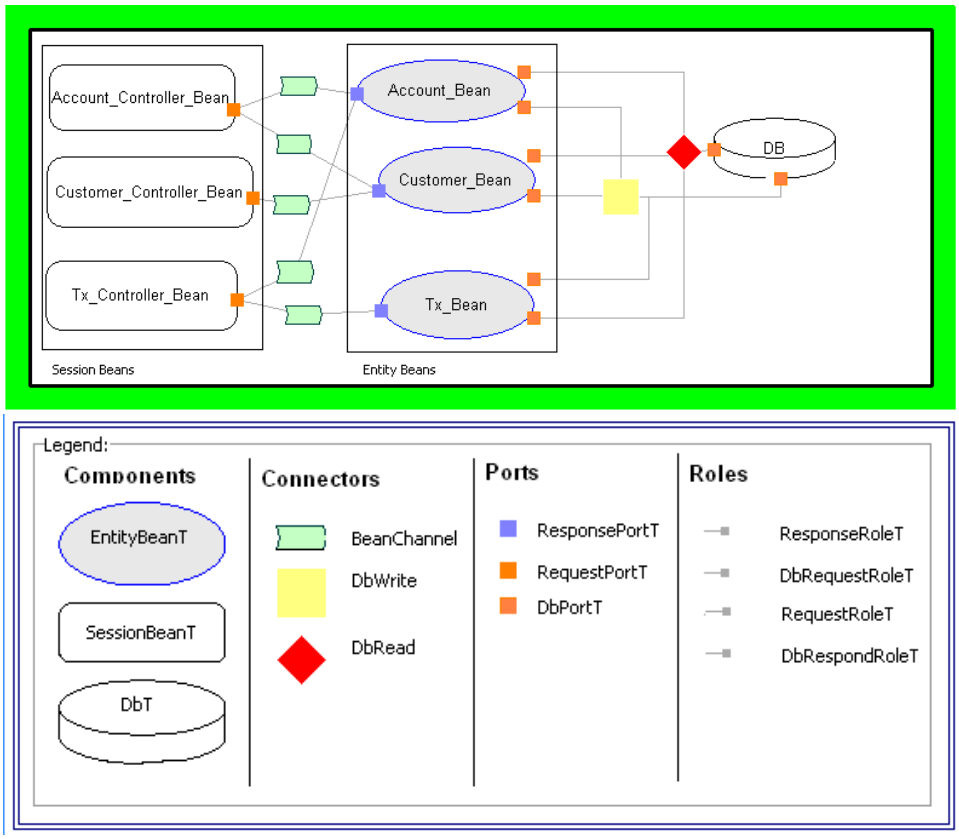


Figure 5.17: Duke's Bank: documented architecture in Acme. The components were added inside the Acme representation of an EJB container (shown as a thick border). Session and Entity Beans are grouped.

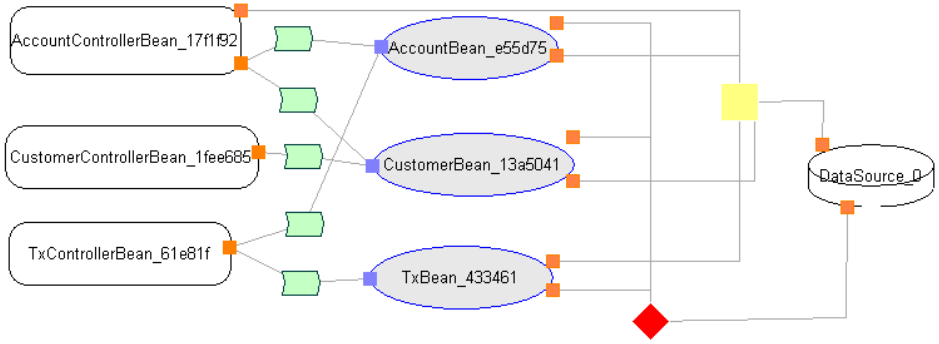


Figure 5.18: Duke's Bank: recovered architecture in Acme.

with around 390 nodes. In this case, the edit script consisted of over 250 renames and over 50 inserts. As expected in this case, THP did not correctly identify the moved view elements.

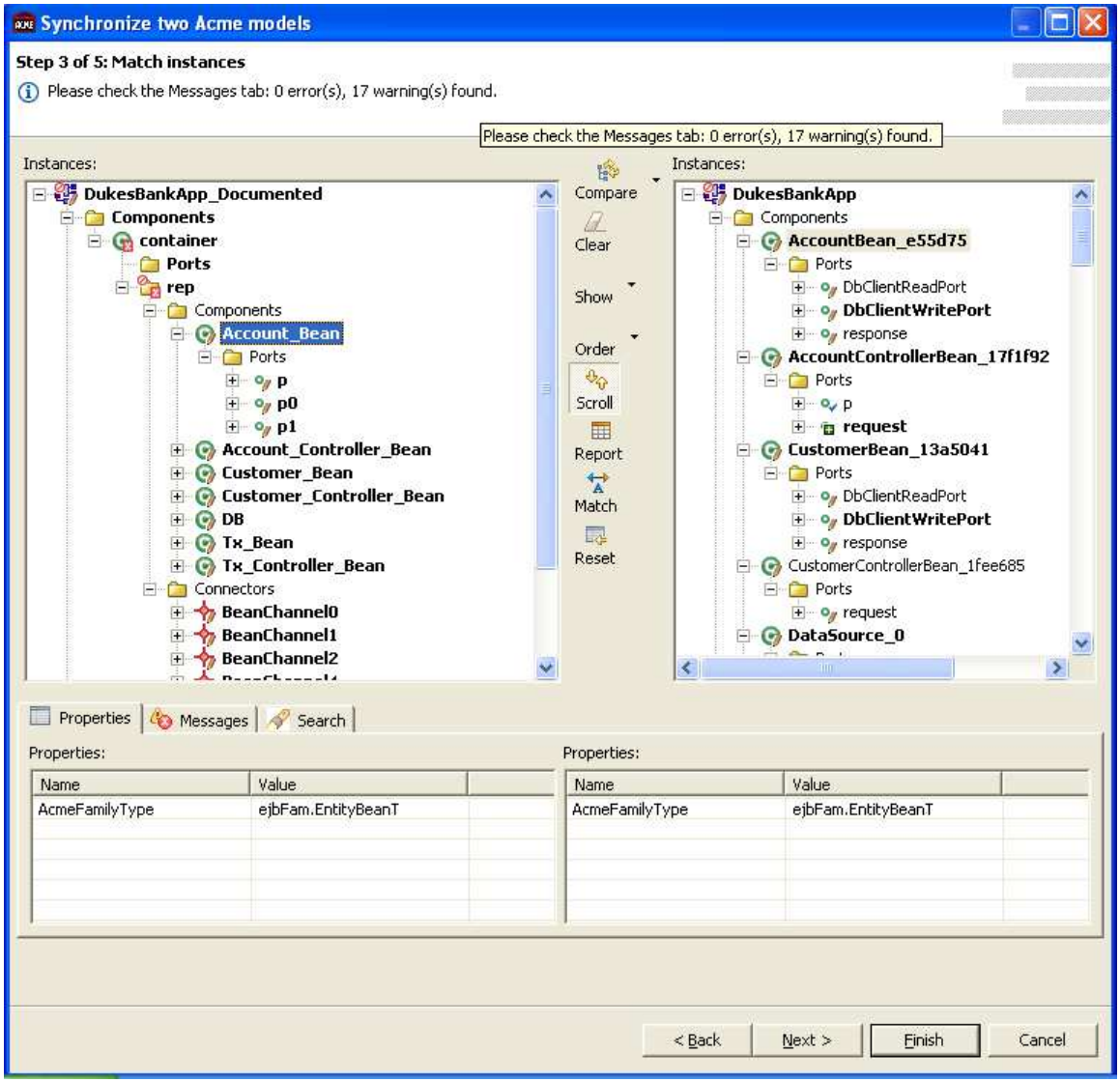


Figure 5.19: Duke's Bank: comparison of the documented and recovered architectures.

5.5.3 Extended Example: HillClimberAJ

In this example, we evaluate the ArchJ2Acme tool again, but this time, we use the feature of allowing the user to force matches. All the examples above actually use the feature to prevent matches, to avoid matching elements of incompatible types.

In previous work, (Abi-Antoun et al. 2007a) re-engineered the original 15,000-line Java implementation into an ArchJava implementation, HillClimberAJ. For this evaluation, we chose HillClimberAJ because it uses a framework, `CIspace`. Indeed, it is common for a product line architecture to use a framework as its platform, and one often needs to compare variants in a product line (Chen et al. 2003). The implementation technology, ArchJava, also made it possible to use a tool to statically extract the built C&C view from the HillClimberAJ code.

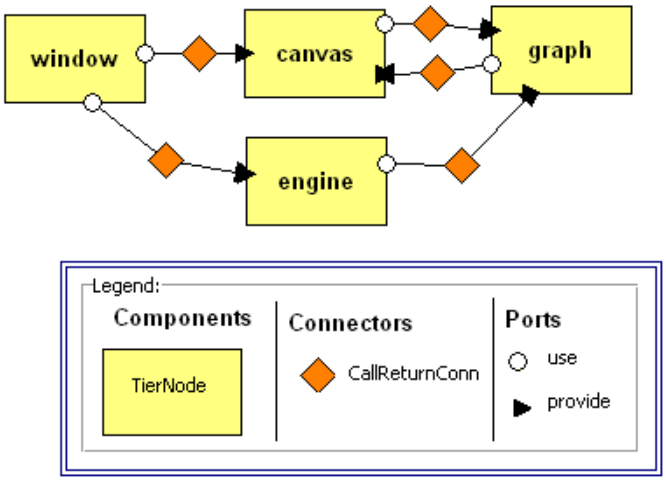


Figure 5.20: HillClimber: Base design for a `CIspace` framework application.

Designed Architecture. The applications that use the `CIspace` framework follow a simple high-level design. An application `window` uses a `canvas` to display `nodes` and `edges` (not shown) of a `graph` in order to demonstrate the algorithms provided by the `engine` (Fig. 5.20).

Built Architecture. We first ran the ArchJ2Acme tool, giving it the designed C&C view (Fig. 5.20), and a C&C view retrieved from the HillClimberAJ ArchJava implementation. In this case, the top-level structure of the designed view was not sufficiently detailed, i.e., the various nodes have roughly the same number of ports. In such cases, structural comparison alone can produce inaccurate results. In this case, ArchJ2Acme incorrectly matched the top-level element `graph` in one view to `window` in the other view.

So the user manually forced the matches between the top-level nodes in the two views, and re-ran the comparison. This time, ArchJ2Acme took into account these manual overrides when matching the instances. Having correctly matched the top-level elements, the comparison highlighted additional differences between the two views. For instance, Fig. 5.21 shows several missing sub-architectures. But the user decided to merge only the changes for the top-level elements and obtained the built architecture in Fig. 5.22.

Discussion. In a product line architecture, each instantiation of a framework often introduces additional runtime dependencies. Indeed, HillClimberAJ added several connections to the documented architecture, and these connections seem mostly justified. For instance, the connection between `engine` and `canvas` is needed since one of the sub-components of `engine` required access to functionality from the `canvas`.

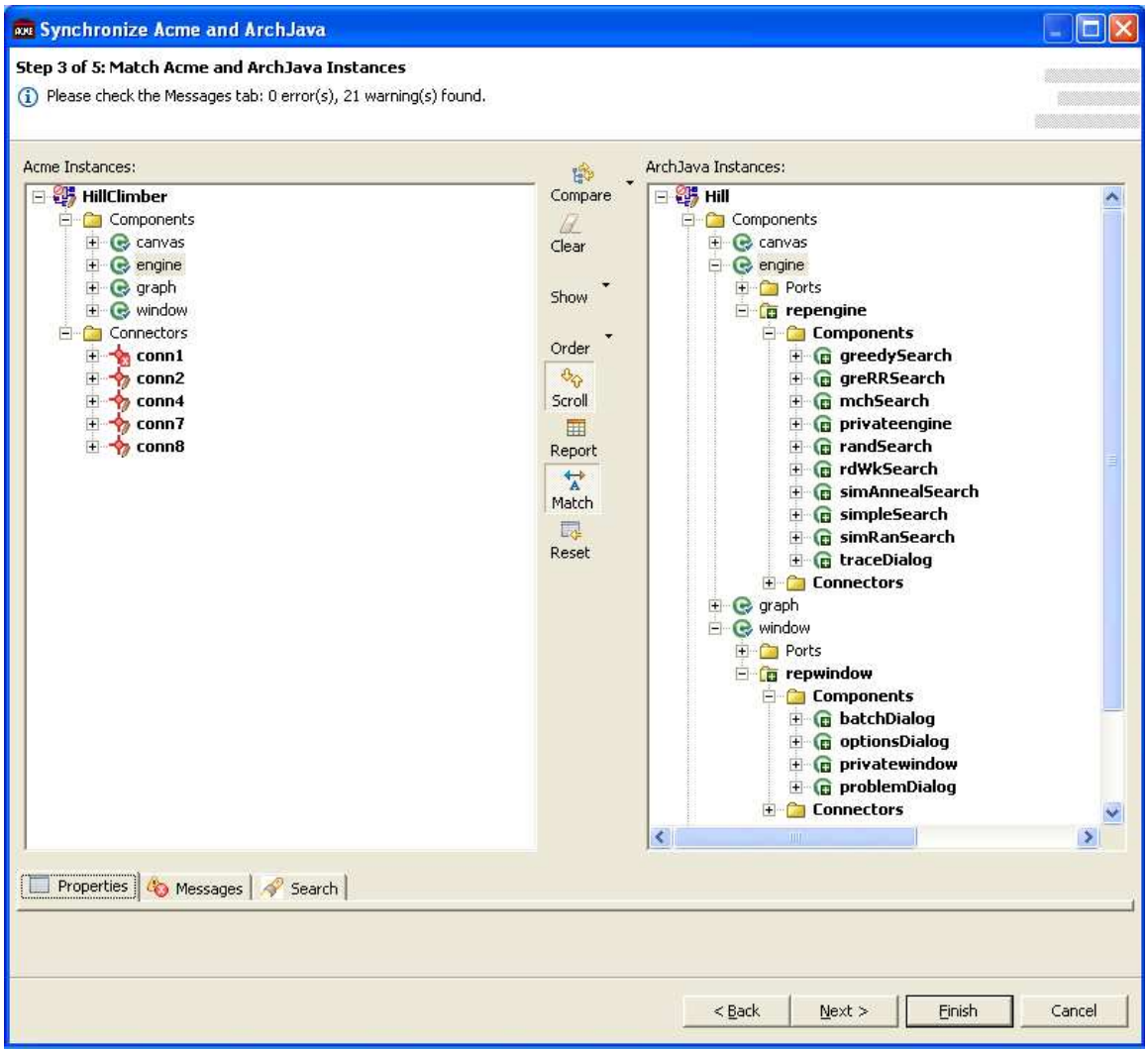


Figure 5.21: HillClimberAJ: manual overrides improve matching the instances. The user forced a match between the `engine` nodes in the two trees by selecting them both and clicking on the 'Match' button before running the differencing algorithm.

5.6 Conclusion

In this chapter, we presented an approach for differencing and merging hierarchical architectural C&C views. We showed how our relaxed assumptions match more closely the problem domain of differencing and merging architectural views after the fact. Finally, we illustrated the tools in extended examples and showed how the approach can find interesting differences in real architectural views.

In this chapter, we used two ArchJava re-implementations of the Aphyds and HillClimber systems, which we referred to as AphydsAJ and HillClimberAJ, respectively. Indeed, a tool can statically extract from ArchJava code a built hierarchical runtime architecture relatively easily, because ArchJava specifies directly in code, architectural hierarchy and instances.

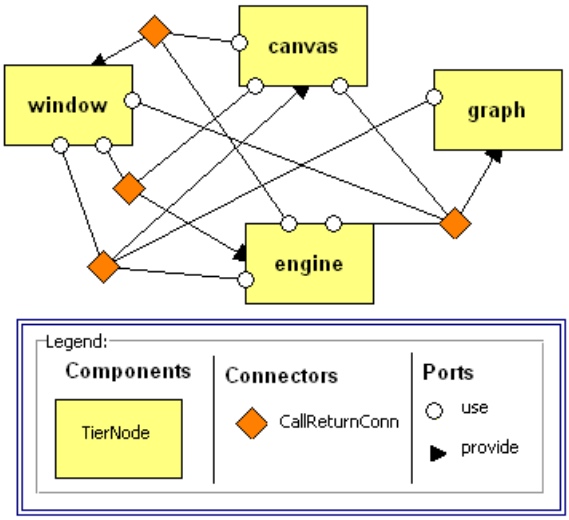


Figure 5.22: HillClimberAJ: built architecture.

In the rest of this dissertation, we revert to the original Java implementations for Aphyds and HillClimber and use SCHOLIA to extract the built architectures from the annotated Java code, and analyze the conformance of the Java implementation, rather than ArchJava, to a target architecture. In the next chapter (Chapter 6), I incorporate the architectural comparison into an end-to-end approach to analyze communication integrity, following the extract-abstract-check strategy.

Credits

Nagi Nahas developed and implemented the tree-to-tree correction algorithm for his M.S. thesis (Nahas 2009), and co-authored the papers on which much of this chapter is based (Abi-Antoun et al. 2006; Abi-Antoun et al. 2008).

Acknowledgements

The author would like to thank Bradley Schmerl for his help with Acme and AcmeStudio, for running DISCOTECT and generating the C&C views for the Duke's Bank case study.

Chapter 6

Conformance Analysis[1]

In this chapter, I demonstrate that SCHOLIA extracts hierarchical object graphs that provide sufficient architectural abstraction to enable conformance analysis. To my knowledge, SCHOLIA is the first static analysis the output of which is readily convertible into a standard hierarchical runtime architecture represented as Component-and-Connector (C&C) view.

I first discuss an analysis to abstract an extracted object graph, and represent it as a standard runtime architecture. I then discuss an analysis which compares the built architecture to a target architecture, analyzes communication integrity in the target architecture, measures their structural conformance, and establishes traceability between the target architecture and the code.

6.1 Introduction

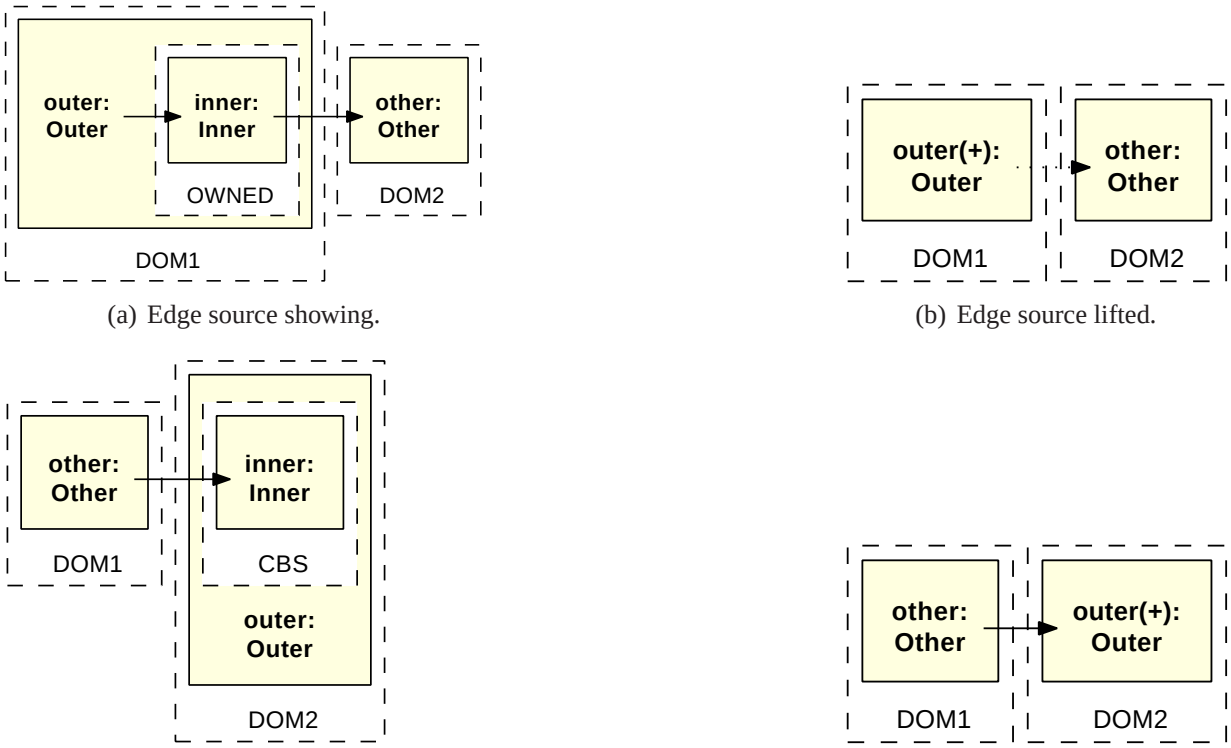
A *designed*, *intended*, *conceptual*, *planned* or *target* architecture is what an architect posits as an abstraction of a system. Even when mostly accurate, such an architecture often omits important communication compared to the *built* or *actual* architecture that an implemented system exhibits[2]. The differences could be omissions in the design or implementation defects. Finding these differences, i.e., analyzing conformance, is an important problem during software evolution (Murphy et al. 2001; Aldrich et al. 2002a). In this dissertation, we deal mostly with architectural structure rather than behavior, so we are concerned with identifying structural differences between a built and an intended architecture.

A designed architecture is often more abstract than the built architecture, but it must still conform to the implementation. SCHOLIA's conformance analysis focuses on communication integrity (Section 1.6, Page 19), i.e., each component in the implementation may only communicate directly with the components to which it is connected in the architecture (Moriconi et al. 1995; Luckham and Vera 1995).

An extracted object graph, however, may not be isomorphic to the architect's intended archi-

[1]Portions of this chapter appeared in (Abi-Antoun and Aldrich 2009b). Preliminary results appeared in (Abi-Antoun and Aldrich 2007c, 2008a).

[2]Throughout this dissertation, we use *built* and *designed* instead of *as-built* and *as-designed* for brevity. The literature refers to these two architectures using many other names, e.g., *concrete*, *implemented* or *physical* for the *built* architecture; and *conceptual*, *idealized* or *logical* for the *designed* architecture (Ducasse and Pollet 2009).



(a) Edge source showing.

(b) Edge source lifted.

(c) Edge target showing.

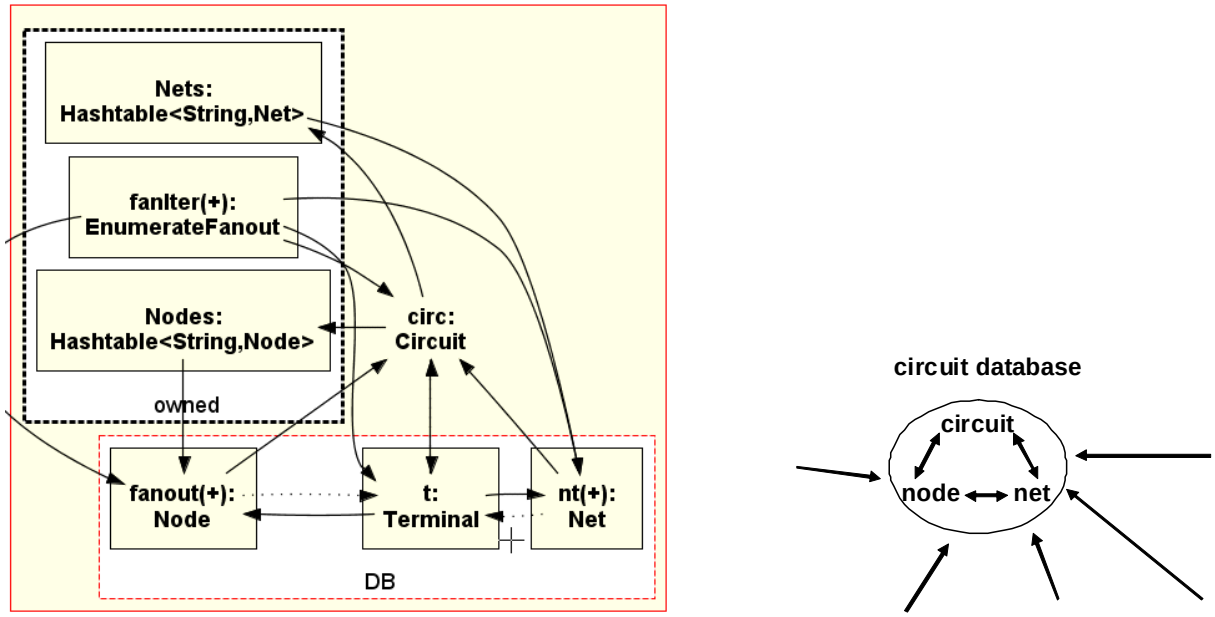
(d) Edge target lifted.

Figure 6.1: Examples of *lifted edges*.

tecture, making it necessary to abstract it further into a built architecture suitable for comparison with the architect's intended architecture. Then, SCHOLIA structurally compares the built and the target architectures and identifies the key differences, using the view synchronization we discussed in Chapter 5. Unlike view synchronization which makes two views identical, the conformance analysis allows a built architecture to contain low-level objects, and does not propagate them directly into the designed architecture. To preserve soundness, however, the analysis still accounts for communication that is not in the designed architecture but occurs in the implementation via these objects.

Finally, SCHOLIA computes conformance metrics to help managers track architectural conformance over time, and derives traceability information that allows an architect to effectively trace architectural violations to the appropriate code.

This chapter weaves the object graph extraction (Chapter 2) and architectural comparison (Chapter 5) into the integrated SCHOLIA conformance analysis, and is organized as follows. In Section 6.2, I discuss how SCHOLIA abstracts an object graph. In Section 6.3, I discuss how SCHOLIA maps a hierarchical object graph to a standard C&C view. In Section 6.4, I discuss the conformance analysis, conformance metrics and traceability support. In Section 6.5, I discuss some of the constraints that could be enforced on the extracted architecture. Finally, I conclude with a discussion in Section 6.6, where in particular, I compare SCHOLIA to other approaches that relate source-level and high-level models.



(a) Object graph.

(b) Target architecture.

Figure 6.2: Aphyds: mismatch between the object graph and the target architecture.

6.2 Abstracting the Object Graph

An extracted object graph provides architectural abstraction by ownership hierarchy and by types. But an object graph may not be isomorphic to an architect's intended architecture, so it may require further abstraction. The steps in the analysis are as follows:

1. Elide and summarize private domains;
2. Skip single domains;
3. Skip objects beyond a certain depth.

We discuss each step in turn, using examples.

Elide and summarize private domains. Object graphs tend to expose the implementation of data structures (O'Callahan 2001, p. 252). SCHOLIA can avoid this problem, when internal state is placed in private domains. In that case, the OOG abstraction step can leverage the semantic distinction between private and public domains, and elide private domains.

For example, in Aphyds, the private domain `OWNED` on `Circuit` stores `Hashtables` of `Node` and `Net` objects (Fig. 6.2(a)), and these objects are not architecturally significant (Fig. 6.2(b)). So the analysis, based on user input, can elide private domains and the objects they contain. To preserve soundness, however, the analysis adds *summary edges* to account for communication through the elided objects. For example, if there is an edge from objects a to b and also from b to c, eliding object b produces a *summary edge* from a to c (Fig. 6.3).

Skip objects beyond a certain depth. The analysis converts an OOG object hierarchy up to a user-selected depth, typically the depth of the hierarchical decomposition in the designed view.

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

Figure 6.3: Example of a *summary edge*.

Reducing the size of the built architecture in this manner speeds up the comparison, but does not affect conformance, because lifted edges account for the elided substructures.

6.3 Describing the Architecture

SCHOLIA can represent the information that it reverse-engineers from the code using different graphical (or non graphical) notations. SCHOLIA uses an architecture description language (ADL) (Medvidovic and Taylor 2000) to represent the built architecture as a standard C&C view (Clements et al. 2003).

There are several benefits to documenting an architecture in an ADL. For example, an ADL can enable various architectural-level analyses. In addition, one could define architectural types, properties and constraints on the architecture to specify architectural intent (Monroe 2001).

For the conformance analysis, SCHOLIA assumes that the designed architecture is represented as a C&C view, instead of an informal diagram. SCHOLIA then compares the built architecture with the intended one.

6.3.1 Architecture description language (ADL)

SCHOLIA uses the following data types from the Acme general purpose ADL (Garlan et al. 2000; Acme 2009) (Section 1.3.2, Page 7).

A Component is a unit of computation and state. A Port is a point of interaction on a Component. A Connector represents an interaction between Ports on Components. A System is a configuration of Components and Connectors. A Component can optionally be decomposed into a nested sub-architecture. A Property is a name and value pair associated with an element. A Group is a named set of Components or Connectors, such as a tier.

As we discussed in Section 5.4 (Page 5.4), the structural comparison uses type information, when available, to improve the match precision. So during mapping, the analysis assigns to the generated C&C elements various types and properties.

For instance, a Port that provides services has type ProvideT, and a Port that uses services has type UseT. The structural comparison uses the type information, when available, to avoid matching a ProvideT Port to a UseT Port, for example.

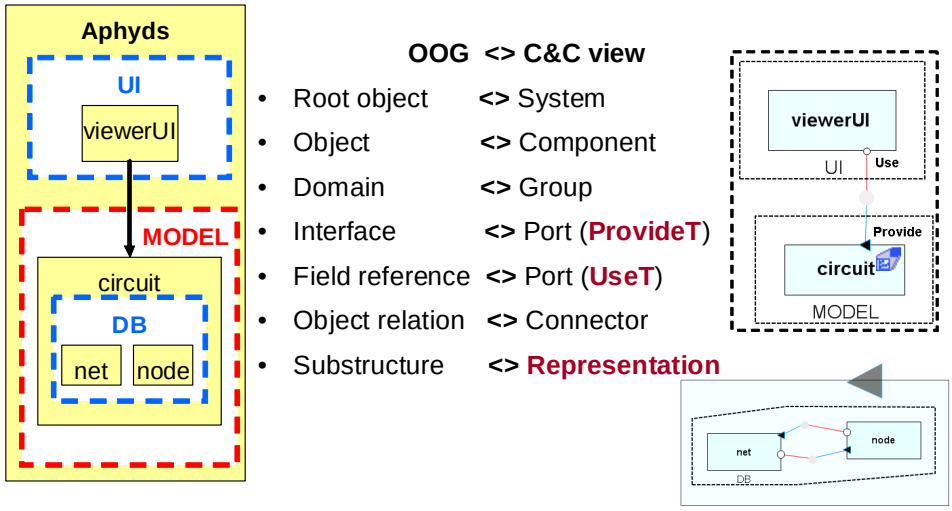


Figure 6.4: Mapping an OOG to a C&C view in the Acme ADL.

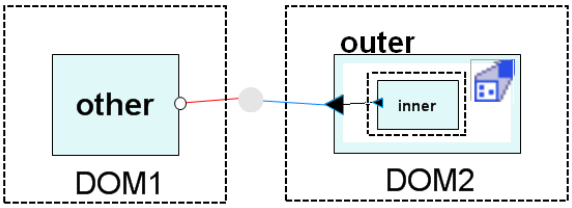


Figure 6.5: A C&C view lifts an edge from component `inner` to component `outer`.

6.3.2 Mapping an OOG to a C&C view

Mapping an OOG to a C&C view works as follows.

Components and Sub-Components. SCHOLIA assumes that an OOG has a single root. So the root object maps to a System. The top-level domains declared by the class of the root object map to the top-level tiers in the System. Each object in the OOG maps to a Component. The OOG hierarchy creates architectural decomposition. If an OOG object declares domains and descendent objects, the corresponding Component has a sub-architecture.

Ports. References between objects create Ports as follows (Fig. 6.4). If object `A` has a field reference of type `T` to object `B`, the corresponding Component A has a Port of type UseT and name B. The Component corresponding to `B` has a Port of type ProvideT and name `T`. And a Connector connects A to B. By default, the analysis does not represent the self-edges in an OOG as connectors in the C&C view, since they are architecturally interesting.

Connection Lifting. The representation of an OOG as a C&C view also lifts edges. Consider an OOG with an edge from `other` to `inner` inside `outer`'s public domain `CBS` (Fig. 6.1(c)).

A C&C view lifts that edge to component outer, shows a connector from other to outer, and a connection from outer to inner (Fig. 6.5).

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

Skip single domains. In an OOG, each object is in a domain, so a systematic conversion of an OOG into a C&C view would create each Component in a Group. Architects typically define tiers only at the top level, and those map to the top-level domains. For example, requiring an Aphyds designed architecture to have a single DB tier inside circuit would be counterintuitive. Unless the developer requests otherwise, the conversion does not create a single tier inside a Component. Unlike eliding private domains, skipping single domains still creates the substructure for those unmapped domains. For example, after eliding the private domain `OWNED` inside `Circuit`, the conversion skips the single public domain DB and creates node and net and the connections between them, directly inside circuit (Fig. 7.13).

Note that although domains play a central role in the annotations, they often disappear after they serve their purpose, which is to distinguish between internal and public state[3]. Recall how in ownership domains, the owner of an object is a domain instead of another object, unlike other ownership type systems (Clarke et al. 1998). Indeed, both public and private domains produce hierarchy in an object graph. But the conformance analysis in SCHOLIA often results in eliding private domains, ending up with a single public domain in a given object, then not representing that domain as a group in the extracted architecture.

6.4 Analyzing Conformance

SCHOLIA can extract the up-to-date built runtime architecture from the code and document it an in ADL. In some cases, a target architecture may be documented, but may be inconsistent with the code. In that case, SCHOLIA can analyze communication integrity in the target architecture.

6.4.1 Conformance Findings

In the terminology of (Murphy et al. 2001), the conformance analysis identifies the following differences between the built and the designed architectures:

- **Convergence:** a node or an edge that *is in both* the built and the designed architectures;

[3]Some type systems embody this idea, and hard-code in each class only one private and one public *boundary* domain (Schäfer and Poetzsch-Heffter 2007).

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

6.4.2 Displaying Conformance

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

6.4.3 Traceability

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

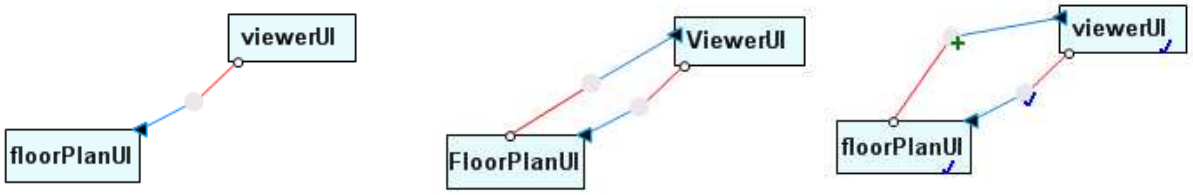
6.4.4 Analyzing Conformance

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

1. Match components, but use the names from the designed view;
2. Highlight differing connections;
3. Summarize divergent components;
4. Check matching substructures recursively.

We discuss each step in turn, using examples.

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



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

Figure 6.6: Displaying a convergence and a divergence.

For Aphyds, the analysis correctly matches built components `ViewerUI` and `FloorPlanUI` to designed component viewerUI and floorplanUI, respectively, but does not rename them (Fig. 6.6).

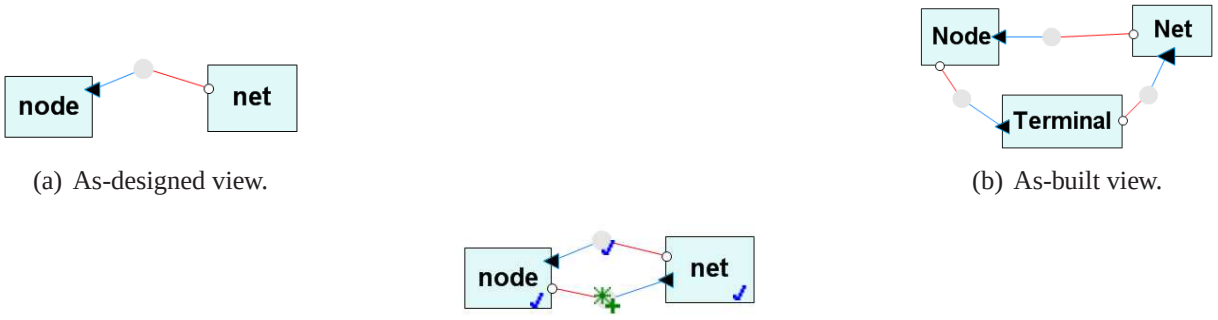
Highlight differing connections. The analysis shows differing connections as divergences or absences. For instance, only the built view has a connector between `FloorPlanUI` and `ViewerUI`, and the latter match the designed components floorplanUI and viewerUI. So the analysis shows a divergent connector from floorplanUI to viewerUI (Fig. 6.6). This requires the following stylized use of ports, which may also make ports easier to understand (Aldrich et al. 2002a).

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

Summarize divergent components. If there are components in the built architecture that are not in the designed architecture, the analysis works differently from view synchronization. Adding these components directly to the designed architecture would clutter it with implementation details. Instead, the analysis accounts for communication in the built architecture that is not in the designed architecture, and adds a *summary connector* to abstract these divergent components and enforce communication integrity.

Consider this other example from Aphyds (Fig. 6.7). In the built view, `Node` connects to `Terminal` and `Terminal` to `Net` (Fig. 6.7(b)). The designed view has node and net, but has no component that matches `Terminal` (Fig. 6.7(a)). The analysis matches node to `Node`, and net to `Net`, respectively. It then shows a divergent connector from node to net, since the designed view does not already have one (Fig. 6.7(c)). If the designed view does have such a connector, the analysis marks it as convergent. Since a summary connector can be either divergent or convergent, the analysis sets a property isSummary on a connector separately from its finding. A decorator overlays the ✳ symbol on a connector when isSummary is set to true.

Viewed differently, the analysis represents using a summary connector any objects in the built



(a) As-designed view.

(b) As-built view.

(c) Conformance view.

Figure 6.7: Displaying a divergence as a *summary connector*.

view that do not have counterparts in the designed view. This allows a designed view to have a coarser granularity of components, and abstract multiple interacting objects with a connector. Indeed, the JavaDoc for Aphyds states as an informal comment in the code that "`Terminal` is a *connection* between a `Node` and a `Net`".

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

Check matching substructures recursively. Designed architectures are often hierarchical, but do not typically have deep hierarchies. An OOG provides architectural abstraction primarily through ownership hierarchy. When an OOG is abstracted into a C&C view—whether restricting the depth of the hierarchy or not, more components in the built C&C view will have substructures than their designed counterparts. To avoid generating many false positives, the analysis ignores the substructures that are in the built view but not in the designed one. Skipping unmatched substructures does not compromise soundness, because both an OOG (Figs. 6.1(b), 6.1(d)) and a built C&C view (Fig. 6.5) lift edges to represent any communication through their substructures.

For instance, viewerUI in the designed view does not define a substructure. So the analysis ignores any substructure in the matching `ViewerUI` in the built view. On the other hand, circuit in the designed view has a substructure including node and net, and matches the `Circuit` object in the built view. In the built view, `Circuit`'s substructure includes the domains `DB` and `OWNED` and the objects inside them. In that case, the analysis recursively examines the substructures of circuit and `Circuit`.

As an aside, had we included private domains when abstracting the Aphyds OOG into a C&C view (as well as generated a Group for the single `DB` domain inside `Circuit`), the analysis would have processed the private domain `OWNED` and generated several undesired divergences. This is because both domains `OWNED` and `DB` are peers in `Circuit`'s substructure (Fig. 6.2), so the analysis cannot ignore the `OWNED` domain and its contents, but process `DB`'s contents.

6.4.5 Measuring Conformance

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

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

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

In terms of face validity, this metric is similar to a *graph edit distance*, which models inconsistencies by transforming one graph into another (Conte et al. 2004). Typical edit operations include the deletion, insertion and relabeling of nodes and edges. Each edit operation is assigned an application-dependent cost. SCHOLIA assigns renames a zero cost and counts insertions (divergences) and deletions (absences).

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

6.5 Enforcing Architectural Structure

Having analyzed conformance and established traceability between the target architecture and the code, we now turn to identifying additional implementation-level violations of the architectural intent. At the code level, we use annotations to enforce local, modular constraints. In addition, we define predicates in the target architecture to enforce global constraints on the runtime architecture.

6.5.1 Code-level constraints

General constraints. In a Java program without ownership domain annotations, changing the runtime architecture is as simple as storing or passing a reference to an object. Adding the

ownership annotations can help enforce some design invariants, for instance, regarding object borrowing, etc. (See observations in Section 4.6.1, Page 128).

Domain links. In addition, it is possible to define various policies in the form of domain links (Section 2.3.2, Page 40).

Limitations. By their nature, domain link annotations are modular, so they cannot express global constraints. In addition, a developer can still add communication pathways by declaring additional domain links or domain parameters and passing additional domain arguments at object allocation sites. Admittedly, code inspections could more closely audit any revision that modifies the domain link annotations. However, since the annotations enforce only modular constraints, it is still necessary to check that code modifications do not adversely impact the intended global architectural structure.

6.5.2 Architectural constraints

Relating the target architecture and the code, together with effective change management, can help detect unwanted architectural violations more effectively than inspecting the program, with or without annotations.

Having an extracted up-to-date built architecture makes it easier to trigger an architectural review. In particular, various constraints can be defined and be enforced on the architecture. Indeed, empirical evidence suggests that such policies are frequently needed during software evolution. For instance, a study using a well-designed framework (JHotDraw) showed that students subverted the framework's design by passing to and storing additional objects in the constructors of classes that implemented the core framework interfaces (Kirk et al. 2006).

In Acme, one can set architectural types, properties and constraints to specify architectural intent (Monroe 2001). By doing so, we may uncover implementation-level violations of architectural types and constraints. In contrast to the modular, code-level constraints, constraints at the architectural level can enforce global constraints on the application structure.

Horizontal conformance. Using predicates to enforce constraints at the architectural level is not novel (Monroe 2001). Indeed, many approaches can check that an architecture obeys a given architectural style, which is similar to verifying *horizontal conformance* (Ducasse and Pollet 2009) between two views at the same level of abstraction. SCHOLIA makes it possible to extract the built runtime architecture, and thus leverage these capabilities. Moreover, since SCHOLIA establishes traceability between the architecture and the code, setting the architectural-level constraints can be used to enforce global constraints on the application structure, within the code.

Architectural types. A built architecture does not usually have rich architectural types. In principle, a developer could enrich the built architecture, abstracted from an object graph, with architectural types. A developer could assign the type individually to the components and connectors in the built view. Or the developer can map implementation types to architectural types, and the tool could automatically assign architectural types to all the components that correspond

to instances of that implementation type. This is only a first approximation, because different instances of the same implementation type such as `HashMap`, could correspond to architectural components of different types. In that case, a developer can still manually fine-tune the automatically assigned types.

One benefit of relating the built and the designed architectures, and enriching the designed architecture with architectural types, is that it can uncover additional violations of the architectural intent in the code.

Analysis-specific properties. It is also possible to define analysis-specific properties and add those to an architecture. For instance, (Abi-Antoun et al. 2007b) defined element-level properties, such as trustLevel, to support an architectural-level analysis to identify spoofing or tampering.

Structural constraints. First-order logic predicates in Acme can enforce various structural constraints (Monroe 2001), such as:

- Component instance c_1 never directly connects to Component instance c_2:
```
forall c1 : Component in self.COMPONENTS |
  forall c2 : Component in self.COMPONENTS |
   connected(c1,c2) ->  !(c1==x AND c2==y);
```
- A Component of type t_1 never directly connects to a Component of type t_2:
```
forall c1 : Component in self.COMPONENTS |
 forall c2 : Component in self.COMPONENTS |
  connected(c1,c2) -> !(declaresType(c1, T1) AND
                        declaresType(c2, T2))
```
- No component in Group g_1 communicates directly with any component in Group g_2:
```
forall g1 in self.GROUPS |
 forall g2 in self.GROUPS |
  forall m1 in g1.MEMBERS |
    forall m2 in g2.MEMBERS |
      m1 = m2 -> g1 = g2;
```

6.6 Discussion

6.6.1 False positives

A designed architecture is often more abstract than the built architecture, but it must still represent all communication that could exist in the implementation. The SCHOLIA conformance analysis enforces communication integrity and ensures that the designed architecture is a conservative abstraction of all the objects in the implemented system and the relations between those objects at runtime. In SCHOLIA, the goal is to have no false negatives in the designed architecture, and show the worst case of possible communication between objects at runtime.

Of course, SCHOLIA, like any other sound static analysis, can generate false positives, and indicate objects or relations that can never exist at runtime, due to infeasible program paths. However, our empirical evaluation in the next chapter will show that SCHOLIA does not generate

many false positives in practice. Moreover, SCHOLIA allows a developer to intervene at different steps in the process, to refine the annotations, control the object graph extraction, tweak the abstraction of the object graph into a C&C view, force or prevent matches during the structural comparison, and select various options when analyzing conformance.

6.6.2 Why an architecture description language?

SCHOLIA uses an architecture description language (ADL) to represent the abstracted object graph and the designed architecture, but SCHOLIA is not tied to any specific notation. For example, SCHOLIA could use UML object diagrams, but UML tools do not support hierarchical object diagrams since they are not part of the UML standard. Alternatively, SCHOLIA could use a UML 2.0 component diagram to describe a runtime architecture (OMG 2008). Future work may consider representing SCHOLIA's output using UML 2.0. Also, there are many other notations that SCHOLIA could potentially use, such as GXL (Holt et al. 2006), an interchange language for many reverse engineering tools.

The main benefit of using an ADL like Acme for SCHOLIA is the ability to declaratively define element types and define properties on those types. When an architectural instance is assigned that type, it automatically inherits those properties. For example, the conformance analysis sets various properties on the elements. In turn, these properties control the display of the elements. Achieving the same effect in many tools requires changing the meta-model used by the tool or using the tool's API to imperatively modify the model. In addition, since AcmeStudio is an Eclipse perspective, all the other tools that SCHOLIA uses are fully integrated around Eclipse. For instance, the CodeTraceJ tool can trace from an architectural element in AcmeStudio to the underlying Java source lines of code, by switching from the AcmeStudio to the Java perspective in Eclipse. Thus, using one of the other Eclipse-based UML tools would not offer additional features.

6.6.3 Why structural comparison?

SCHOLIA compares the designed and the built architectures using a structural comparison that works with hierarchical views, does not assume unique identifiers, detects renames and allows forcing or preventing matches between selected view elements. These assumptions closely match the problem of analyzing conformance after the fact. SCHOLIA does not assume that the architectural components have unique identifiers, which would simplify the graph comparison considerably (Conte et al. 2004). The main benefit of using structural comparison enables SCHOLIA to detect renames between the built and the designed architectures, which can partly occur because of the way we extract an OOG.

The OOG extraction nondeterministically selects a label for a given object o based on the name or the type of one of the references in the program that points to o. Thus, detecting renames ensures a developer can still rename fields or local variables or types in the program without impacting conformance. Avoiding the rename problem entirely would require additional annotations to specify in code the preferred labels. But we prefer to keep the design of the annotations minimal, and focused on just specifying object encapsulation and logical containment properties.

6.6.4 Relation to Reflexion Models

I modeled SCHOLIA closely after Reflexion Models (Murphy et al. 2001), a standard-bearer in analyzing the conformance of the code architecture, which I refer to as RM. To be clear, RM cannot handle the runtime architecture. However, drawing the similarities between RM and SCHOLIA more explicitly is informative, because both approaches analyze conformance using the extract-abstract-check strategy. Also, to my knowledge, other static conformance checking techniques of the code architecture have comparable expressiveness to RM.

RM works as follows. In RM, a third-party tool extracts a *source model* from the source code. The RM user posits a designed *high-level model* and a *map* between the source and high-level models. RM pushes each interaction described in the source model through the map to infer edges between high-level model entities. RM then compares the inferred edges with the edges stated in the high-level model and shows the differences. A developer can iteratively: (a) modify the high-level model; (b) modify the source model; (c) modify the map; (d) trace a conformance finding to code; and (e) optionally, change the code to conform to the architecture.

In SCHOLIA, the *source model* is the source code with the ownership domain annotations. The target architecture is the *high-level model*. And the structural comparison, together with optional manual input to force or prevent matches, produces the *mapping*. Similarly to RM, a SCHOLIA user can iteratively modify the designed architecture, the annotations in the program, the structural comparison or the mapping, and trace from the conformance view to the code.

RM does not extract a complete abstraction to avoid obtaining a model that developers do not recognize. In SCHOLIA, the OOG represents a complete model, but developer-specified annotations help obtain meaningful abstractions. In SCHOLIA, an extracted OOG is incomplete only if the program is not completely annotated, there are remaining annotation warnings, or the virtual field annotations do not soundly summarize all the external entities.

In RM, if the mapping generates a node that is not the designed view, RM automatically adds it to the designed view, i.e., RM has no divergent or absent nodes. As a result, RM need not summarize entities that are present in an implementation with an edge in the high-level model. RM also assumes that node names have exactly matching information (names and token types) and uses a graph connection model without ports on nodes. When RM compares the inferred edges with the edges stated in the high-level model, it produces simple metrics based on the number of divergences, convergences and absences. RM measures unmapped entries of the source model, which gives an indication of the incompleteness of the mapping, which is somewhat similar to our WARN metric.

In the base RM technique, the end-user manually generates the map. Building a map from scratch is a laborious process. For example, for an 1-MLOC system, the map had over 1,000 entries (Murphy et al. 2001). To alleviate the burden of producing a map manually, (Christl et al. 2005) proposed semi-automated clustering algorithms to obtain a mapping. Their evaluation on various code architectures determined that the engineer may spend significant effort deriving a good partial mapping, including fine-tuning the clustering algorithm's parameters.

Producing the RM mapping file appears more straightforward than adding ownership annotations, but it is not amenable to type inference. Furthermore, since RM is used for the module view, the mapping need not take into account inheritance or possible aliasing in object-oriented code. These are non-issues in a module view, but of course, are very important in a runtime view

(Section 6.6.5, below).

In the base RM technique, both the high-level model and the map are non-hierarchical. (Koschke and Simon 2003) proposed hierarchical extensions that account for substructure by lifting edges. SCHOLIA considers both the designed and the built architectures as hierarchical.

6.6.5 Mapping Code to High-Level Models

I will now explain how SCHOLIA generalizes previous techniques for extracting an abstraction of the runtime structure of source entities, and why SCHOLIA's more sophisticated source abstraction technique is needed to analyze conformance of the runtime architecture.

Let us hypothetically attempt to apply RM to a runtime architecture to better understand why an approach that works on the code architecture cannot handle the runtime architecture. In particular, let us map source entities to a hierarchical runtime structure.

Let us assume that RM supports tiers and qualify a component by its tier using the :: symbol. The Java RM tool, JRM (JRM 2003), can map class `Circuit` to a circuit high-level element, as follows:

```
class Circuit to MODEL::circuit.
```

A class is a code entity, not a runtime entity. The above map entry could indicate that it is mapping all the instances of the `Circuit` class to a `circuit` element in a code architecture. But in an object-oriented system, there is usually more than one instance of many classes, and each instance can map to a different component in a runtime architecture.

When working with a runtime architecture, the source model must reflect the *runtime structure* of the system and represent an object graph. Instead of mapping a class or all of its instances, we need to map runtime objects. A static analysis knows only about field or variable declarations in the program, which denote references to runtime objects. We use `Main.circuit` to denote a `circuit` field declared in class `Main`, which points to an instance of the `Circuit` class at runtime. Assume we extend JRM and map:

```
object Main.circuit to MODEL::circuit.  // circuit has type Circuit
```

In object-oriented code, multiple code elements could correspond to the same object at runtime. An architecture would be deceptive if it showed one runtime entity as two components. For instance, an architectural security analysis could assign one runtime entity two different values for a key trustLevel property. So, all the references that may alias, i.e., refer to the same object at runtime must map to the same component in the architecture. For example, a reference of type `Circuit` and another of type `ICircuit`, an interface that `Circuit` implements, may alias, so we somehow also have to map both to the same runtime component.

```
object Main.iCircuit to MODEL::circuit.  // iCircuit has type ICircuit
```

In addition, one code entity can map to multiple components in a runtime architecture. A code architecture such as a class diagram would show one `Vector` *class*, and `Node` and `Net` *classes* that have a module dependency on `Vector`. In a runtime architecture, different instances of `Vector` are often part of conceptually different components. For instance, a `Node` object

has a `Vector` object of `Terminal` objects. Another distinct `Vector` object, also of `Terminal` objects, is part of a `Net` object. To support this feature, a previous system (LR) defines a context parameter using an annotation in the code, and binds that parameter to different actual contexts using additional annotations (Lam and Rinard 2003). But in LR, all the context parameters bind transitively to a top-level context such as MODEL. As a result, LR extracts a non-hierarchical model.

Most ADLs support the hierarchical decomposition of a component into a nested sub-architecture (Medvidovic and Taylor 2000). For example, the Aphyds designed architecture shows node and net inside circuit's sub-structure (Fig. 1.1). So first, we define a sub-structure, i.e., a nested context or tier DB, inside circuit, which we refer to as if it were a field, as in circuit.DB. We then model objects such as `node` and `net` as being *part of* a `circuit` object, by mapping them to components inside DB. For example, we map a field `node` to a node component in circuit's `DB`:

```
object Circuit.node to MODEL::circuit.DB::node.
```

SCHOLIA is more expressive than LR, since it can bind a context parameter to a nested context such as a circuit.DB, thus achieving hierarchy. Also, SCHOLIA is more expressive than RM. By binding different domain parameters to the same actual domains, the analysis can map multiple code elements to the same architectural component. Similarly, by binding one domain parameter to different actual domains, the analysis can also map one code element to multiple architectural components.

In summary, the *implicit* map generated by SCHOLIA generalizes previous maps (Murphy et al. 2001; Lam and Rinard 2003), accounts for inheritance and aliasing, and relates a rich, hierarchical description of an architect's intended runtime architecture to a hierarchical representation of the runtime structure of source code entities.

6.7 Summary

This chapter weaves the architectural extraction (Chapter 2) and architectural comparison (Chapter 5) into the SCHOLIA end-to-end architectural conformance approach.

I discussed an analysis that takes a hierarchical object graph, extracted statically, abstracts it into a built runtime architecture represented as a C&C view, to make it comparable to a target architecture. I also discussed how the conformance analysis enforces communication integrity in the designed architecture, while allowing a built architecture to contain low-level objects, without propagating them directly into the designed architecture.

To my knowledge, SCHOLIA is the first approach to extract statically a runtime architecture from a program in a widely used object-oriented language, using annotations. If an intended architecture exists, SCHOLIA can analyze, also at compile time, communication integrity between the code and the target architecture. Finally, SCHOLIA can establish traceability between an implementation and an intended runtime architecture.

In the next chapter (Chapter 7), I evaluate SCHOLIA on several real representative object-oriented systems, and show that, in practice, SCHOLIA can find interesting structural differences between an existing system and its target runtime architecture. The evaluation will confirm what

others have reported (Murphy et al. 2001; Aldrich et al. 2002a), that informal diagrams often omit important communication. Thus, analyzing conformance after the fact can be very useful during software evolution to ensure that architects base their important decisions on accurate architectures.

Acknowledgements

The author would like to thank Bradley Schmerl for his help with Acme and AcmeStudio. In addition to the thesis committee, David Garlan, Mary Shaw and Larry Maccherone gave us helpful comments on the approach.

Chapter 7

Evaluation of the Conformance Analysis[1]

To demonstrate that SCHOLIA works in practice, I evaluate the end-to-end approach on several real representative object-oriented systems. In this chapter, I demonstrate that, in practice, SCHOLIA can be applied to existing systems while adding or refining only annotations in the code, that SCHOLIA can find interesting architectural violations, that these violations can be traced to code, and that SCHOLIA computes sensible metrics.

7.1 Introduction

Before I discuss the individual case studies, I discuss the research questions I wanted the evaluation to answer (Section 7.2), the tool support that I built to evaluate SCHOLIA (Section 7.3) and the evaluation methodology (Section 7.4). I then present four extended examples: Aphyds (Section 7.5), JHotDraw (Section 7.6), HillClimber (Section 7.7), and CryptoDB (Section 7.8). I conclude this chapter with a discussion (Section 7.9).

7.2 Research Questions

Our evaluation aims to answer the following hypotheses (Section 1.10, Page 25):

> *H-4: An analysis can abstract an object graph into a built component-and-connector runtime architecture represented in a standard architecture description language.*
>
> *H-6: An analysis can check communication integrity with a target architecture, establish traceability between the target architecture and the code, and compute structural conformance metrics in practice.*

We refine the hypotheses into the following research questions:

RQ3 – Conformance: *Can the conformance analysis display a meaningful conformance view, enable tracing a finding to the code, and compute sensible conformance metrics?* The measurable criteria are: few false positives, a readable conformance view that does not have so many divergences that it is almost a fully connected graph, and the ability to trace to the right

[1] Portions of this chapter appeared in (Abi-Antoun and Aldrich 2009b).

code locations. Ideally, the goal is to minimize the number of divergences and absences that the tool reports, or to ensure that they correspond to cases where the implementation violates the architectural intent.

RQ4 – Value: *Can* SCHOLIA *identify interesting structural differences between built and designed architectures in real systems?*

The conformance analysis requires the architectural extraction (Chapter 2) and the architectural synchronization (Chapter 5). We also reuse two of the subject systems from Chapter 4, JHotDraw (Section 7.6) and HillClimber (Section 7.7), on which we previously evaluated the annotations and the object graph extraction (See Sections 4.6, 4.7). This chapter also presents an end-to-end evaluation on two new subject systems, Aphyds (Section 7.5) and CryptoDB (Section 7.8). So we revisit the corresponding hypotheses and research questions below:

> *H-1: Lightweight typecheckable ownership annotations can specify, within the code, local hints about object encapsulation, logical containment and architectural tiers.*
>
> *H-2: In practice, a static analysis can extract from an annotated program a global, hierarchical object graph that provides architectural abstraction by ownership hierarchy and by types.*
>
> *H-5: An analysis can structurally compare the built architecture to a documented target runtime architecture.*

We refine the above hypotheses into the following research questions:

RQ1 – Extraction: *Can* SCHOLIA *extract statically a meaningful, hierarchical, Component-and-Connector (C&C) runtime architecture?* The measurable criteria here are to abstract away low-level objects that are implementation details.

RQ2 – Comparison: *Can the structural comparison meaningfully compare a built architecture extracted from the implementation to a designed architecture?* The measurable criteria here are to minimize the occurrences where a developer must manually force or prevent matches between the view elements.

7.3 Tool Support

I developed several tools and integrated them with existing tools (Fig. 7.1), in order to support the SCHOLIA approach (Section 1.7, Page 20):

- **AcmeStudio** (Schmerl and Garlan 2004; AcmeStudio 2009) is a modeling environment for Acme. In SCHOLIA, a developer uses AcmeStudio to document the designed architecture, display the extracted built architecture, and display the conformance view. I used AcmeStudio to generate all of the C&C views in this document.
- **ArchCheckJ:** ArchCheckJ (stands for Architectural annotation Checker for Java) typechecks the annotations added to the code and is discussed in Section 4.3.1 (Page 122);
- **ArchRecJ:** ArchRecJ (stands for Architectural Recovery for Java) extracts an OOG from annotated code and is discussed in Section 4.3.2 (Page 122);
- **ArchCog:** ArchCog (stands for Architectural Component Object Graph) abstracts an OOG into a C&C view, using the techniques I discussed in Section 6.2 (Page 207). A developer

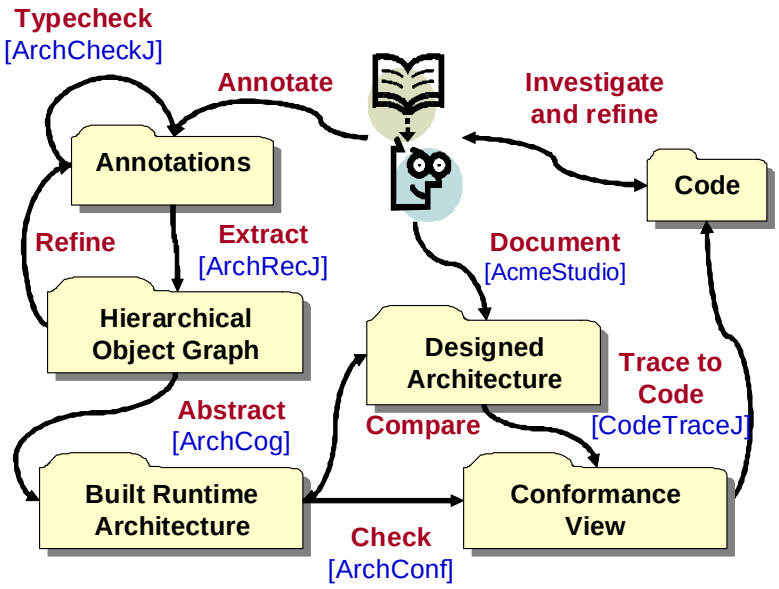


Figure 7.1: Tools to support the SCHOLIA approach.

can also soundly elide private domains. Finally, a developer can restrict the unfolding depth at which to represent the C&C view;

- **ArchConf:** ArchConf (stands for Architectural Conformance analyzer) analyzes the conformance between two C&C views, generates a *conformance view* and computes the metrics, as discussed in Section 6.4 (Page 210). ArchConf allows a developer to confirm the results of the structural comparison, or to manually force or prevent matches and rerun the comparison;
- **CodeTraceJ:** CodeTraceJ loads the traceability of an element in the conformance view, opens the corresponding source files and highlights the appropriate lines. Because AcmeStudio is implemented as an Eclipse perspective, CodeTraceJ allows a developer to trace seamlessly from a conformance view in AcmeStudio to the Java code in the Eclipse JDT.
- **ArchMod:** ArchMod (stands for Architectural Modification wizard) modifies the original designed architecture, by taking a divergent element from the conformance view and adding it to the designed view, or deleting an absent element from the designed view.

7.3.1 ArchCog

The details of abstracting an object graph into a C&C view were discussed previously (Section 6.2, Page 207). The object graph abstraction tool, ArchCog, offers the following features (Figs. 7.2, 7.3):

- **Control the unfolding depth:** the developer can control the depth of the OOG from which to generate the C&C view;
- **Elide private domains:** the developer can make the tool soundly summarize private domains (this is the default);
- **Skip singleton domains:** a developer can make the tool not generate an Acme Group for

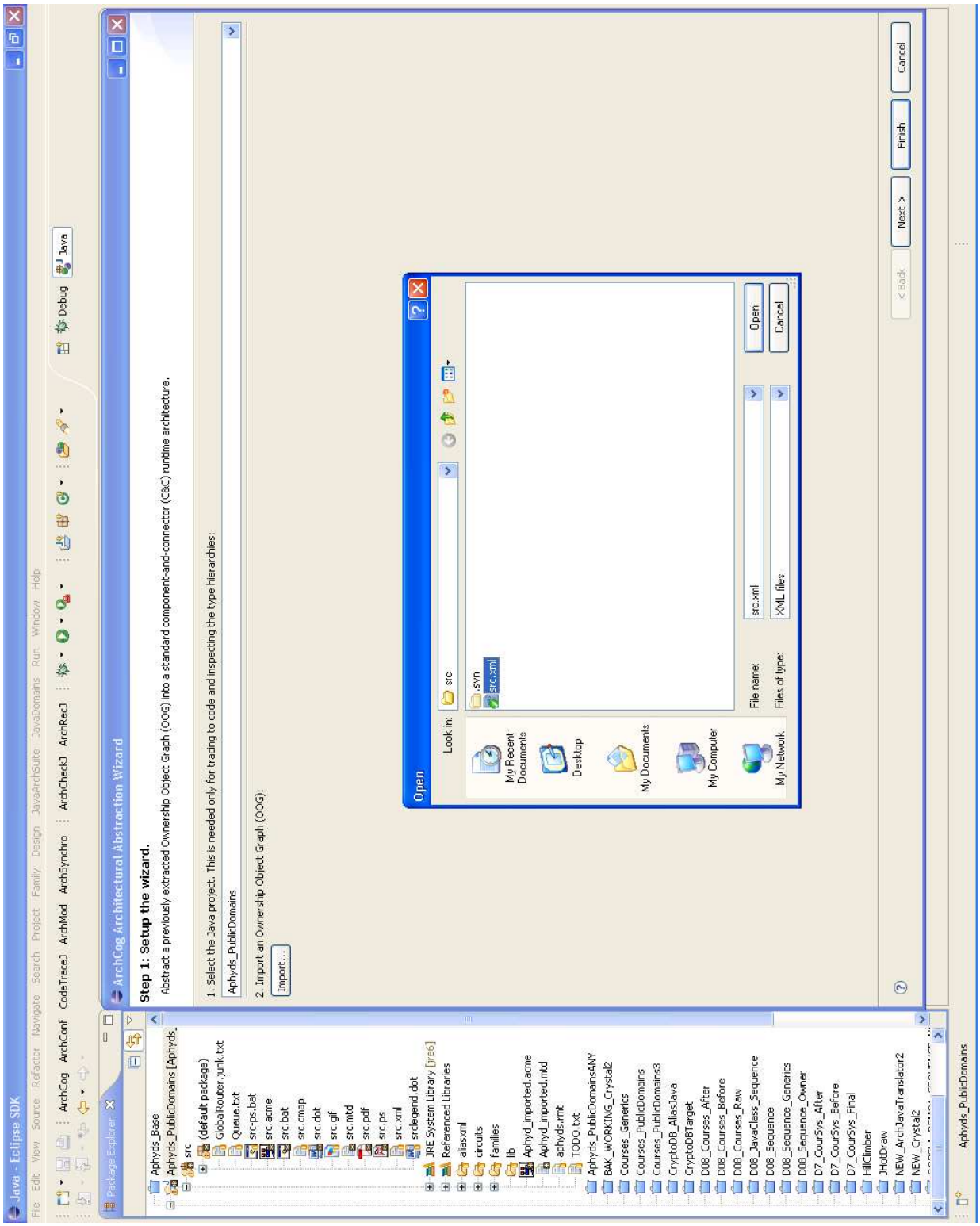


Figure 7.2: ArchCog tool. Step 1: select the project and the OOG.

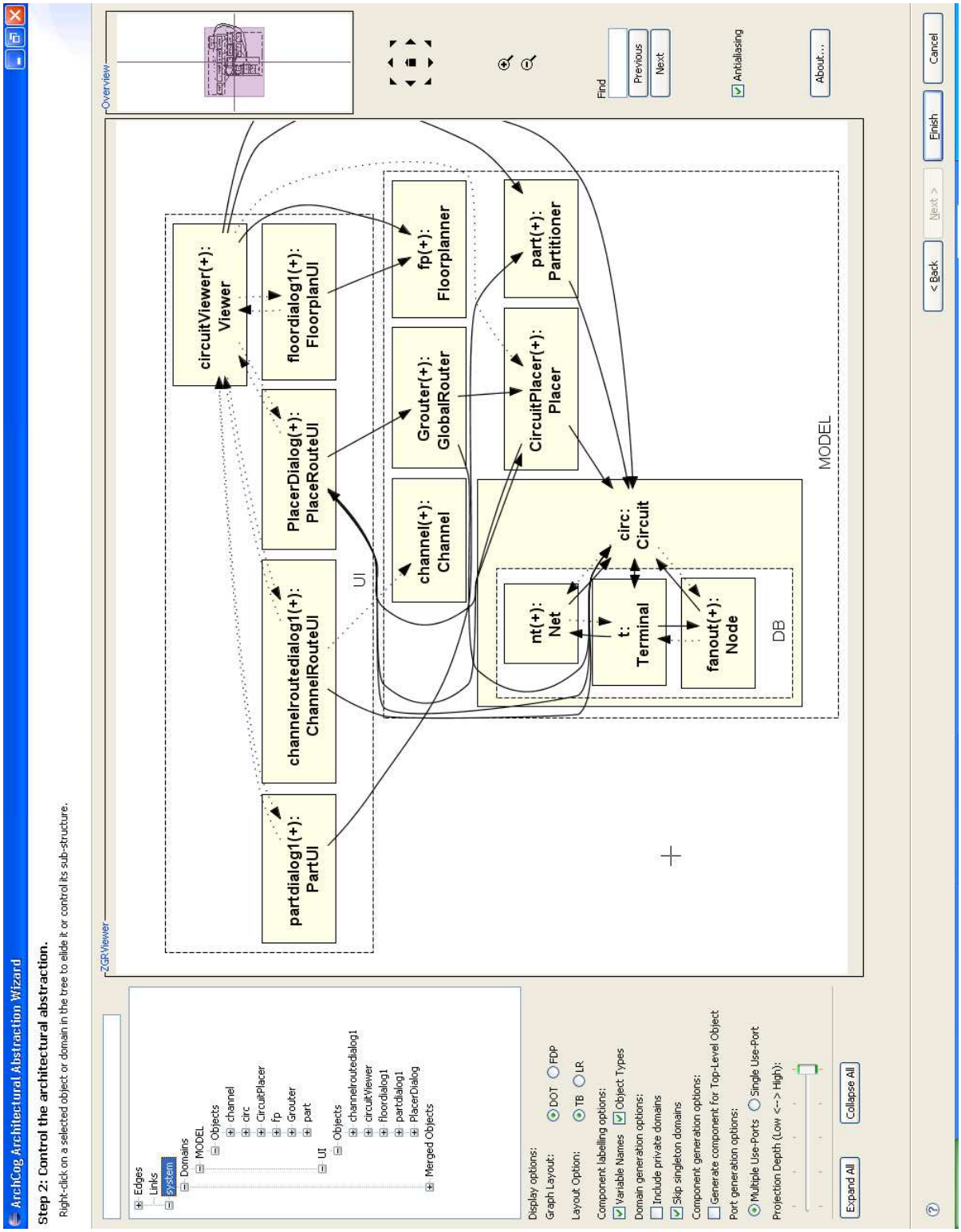


Figure 7.3: ArchCog tool. Step 2: control the abstraction options.

a singleton domain (this is the default);
- **Set component labeling:** the developer can choose various labeling options for labeling the C&C view elements.
- **Persist abstracted OOG:** ArchCog persists an abstracted OOG into an Acme file.

7.3.2 ArchConf

The details of analyzing conformance were discussed previously (Section 6.4, Page 210). The conformance analysis tool, ArchConf, offers the following features (Figs. 7.4, 7.5, 7.6):

- **Step 1: select the built and the target architectures.** The developer selects the built and the target C&C views (Fig. 7.4);
- **Step 2: compare the built and the target architectures.** The developer compares the built and the target C&C views (Fig. 7.5). Typically, the developer simply confirms the comparison results;
- **Step 3: compute the conformance view.** The tool generates the conformance view in Acme (Fig. 7.6).

7.3.3 CodeTraceJ

The CodeTraceJ tool does not currently have a user interface. The functionality is invoked through a menu item that is added to Eclipse. First, CodeTraceJ loads the traceability of an element from the conformance view, where it is stored in properties in the Acme model. Then, it asks Eclipse to load the corresponding Java source files. Finally, CodeTraceJ uses built-in Eclipse functionality to highlight the appropriate lines.

7.3.4 ArchMod

The ArchMod tool does not currently have a user interface. The functionality is invoked through menu items that is added to Eclipse. Because the conformance view is a copy of the target architecture, ArchMod modifies the designed architecture directly. In future work, we will enhance the user interface for ArchMod, to enable a developer to preview the changes before they are applied to the designed architecture.

7.4 Evaluation Methodology

During the evaluation, the experimenter follows closely the SCHOLIA methodology, and uses the tools in Section 7.3 as follows.

In Eclipse, the experimenter uses the AcmeStudio perspective to document the designed architecture in the Acme architecture description language. Still in Eclipse, he switches to the Java development perspective, loads the implementation project, adds ownership domain annotations to the code as Java 1.5 annotations, and invokes the ArchCheckJ typechecker. He double-clicks on a warning in the Eclipse problem window to go to the offending line of code, and attempts to address the relevant annotation warnings.

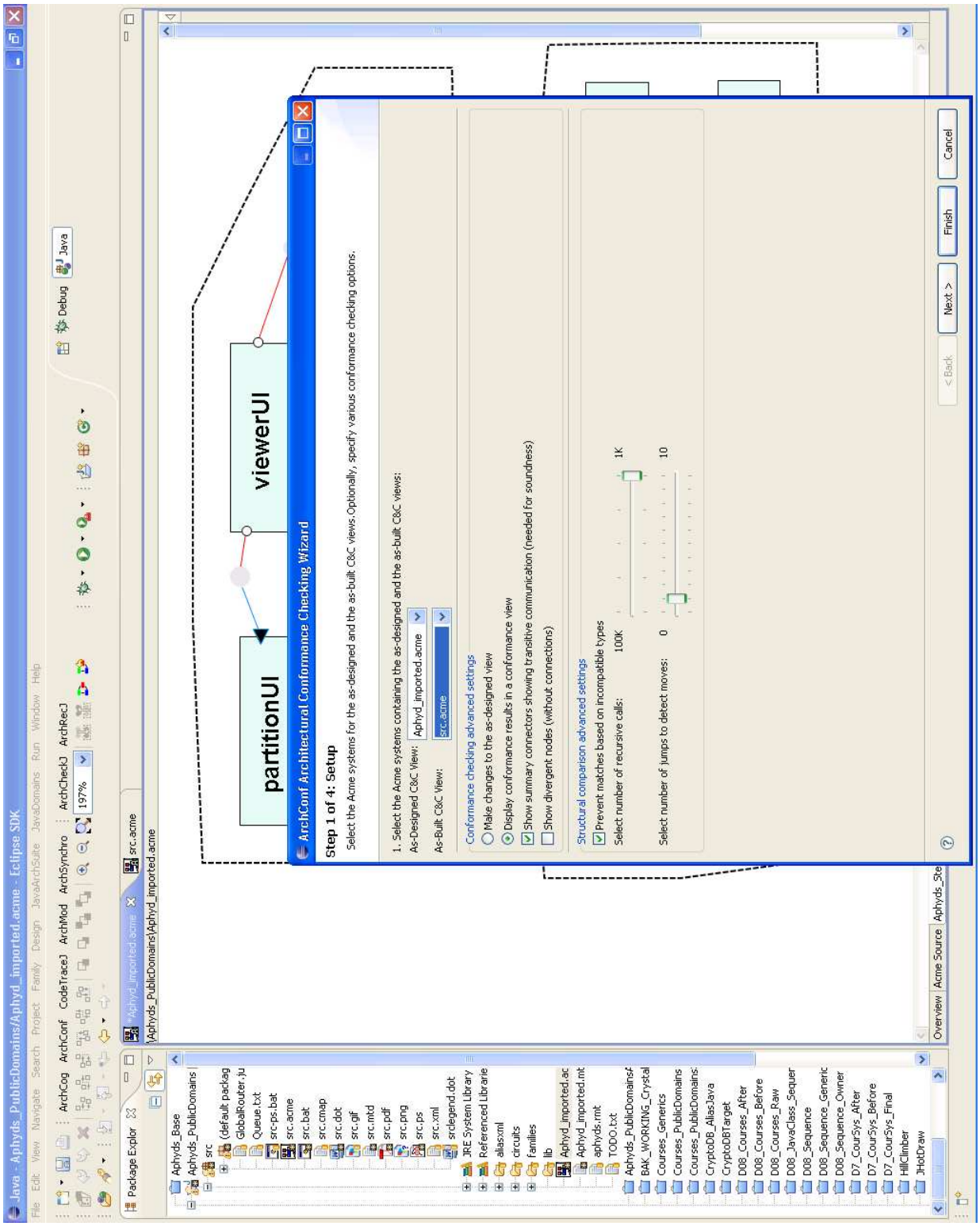


Figure 7.4: ArchConf tool. Step 1: select the built and the target architectures.

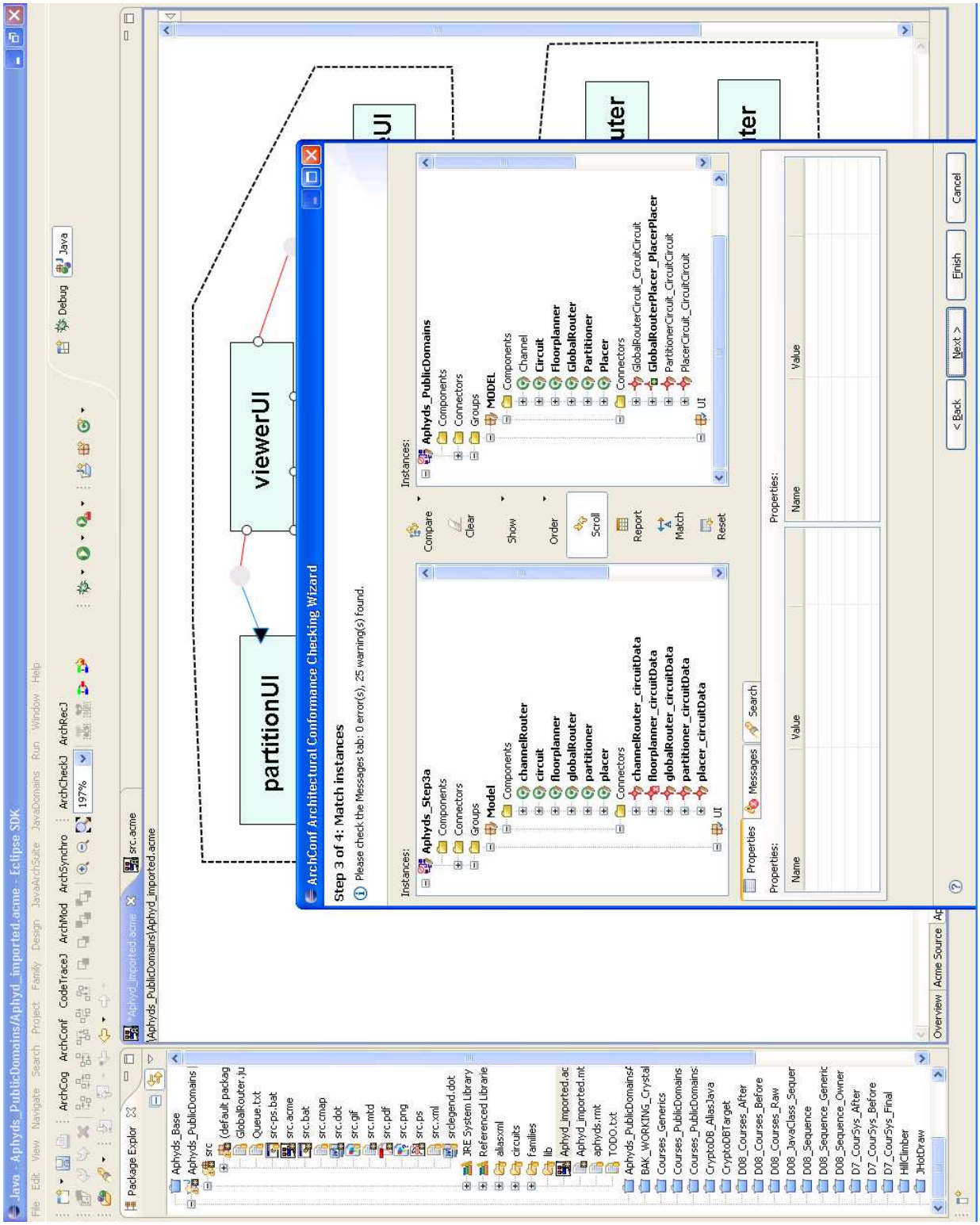


Figure 7.5: ArchConf tool. Step 2: compare the built and the target architectures and examine the results of the structural comparison.

Figure 7.6: ArchConf tool. Step 3: compute the conformance view.

After the developer annotates most of the program and eliminates most of the serious warnings, he uses the ArchRecJ tool to extract an object graph. For a meaningful comparison, the extracted object graph (which will be abstracted into the built architecture), and the target architecture must have similar tiers, similar hierarchical decomposition, and similar components and tiers at each hierarchy level. Typically, the experimenter spends some time refining the annotations and visualizing object graphs, until the extracted object graph is roughly comparable to the designed architecture.

He then invokes the ArchCog tool to abstract the extracted object graph into a built architecture represented as a C&C view in AcmeStudio. He then points the ArchConf tool to the extracted built architecture and to the designed architecture and examines the results of the structural comparison. ArchConf displays the two views side-by-side in a tree-form and shows the mapping by overlaying icons on the affected elements in each tree (Fig. 7.5). If an element is matched or renamed, the tool automatically selects and highlights the matching element in the other tree; for inserted or deleted elements, the tool automatically navigates up the tree until it reaches a matched ancestor.

In ArchConf, the developer typically accepts the comparison results. But if the comparison mismatches some elements, he can manually force or prevent matches between those elements, and rerun the comparison. Once the developer accepts the comparison results, ArchConf then creates a conformance view of the designed architecture, and displays the conformance metrics in an output window.

Back in the AcmeStudio perspective, the developer examines the conformance view, and investigates unexpected divergences. He uses CodeTraceJ to confirm a convergence or trace a divergence to the code. If the divergence is critical, he may modify the implementation to eliminate the architectural violation. Alternatively, he may update the designed architecture if he considers that the conformance analysis highlights an error or omission in the architecture. He can update the design architecture either manually, or he can use ArchMod to propagate components or connectors from the conformance view back to the target architecture.

7.5 Extended Example: Aphyds

We now describe analyzing the conformance of the Aphyds system using the SCHOLIA methodology and tools. The experimenter (hereafter "I") developed several of the tools, but none of the subject systems.

The process was iterative as a whole, and involved both macro- and micro-iterations. A macro-iteration consists of documenting the designed architecture, adding the annotations, extracting an OOG, abstracting it into a built C&C view, and analyzing its conformance. A micro-iteration can consist of the process of iterating the annotations and extracting OOGs before converting an OOG into a C&C view. One exits a micro-iteration when an OOG has a reasonable abstraction level, and no longer shows low-level objects such as `Vectors` in the top-level domains. Retrospectively, we present the Aphyds evaluation as two macro-iterations and show the evolution of the conformance metrics across the two macro-iterations (Table 7.1).

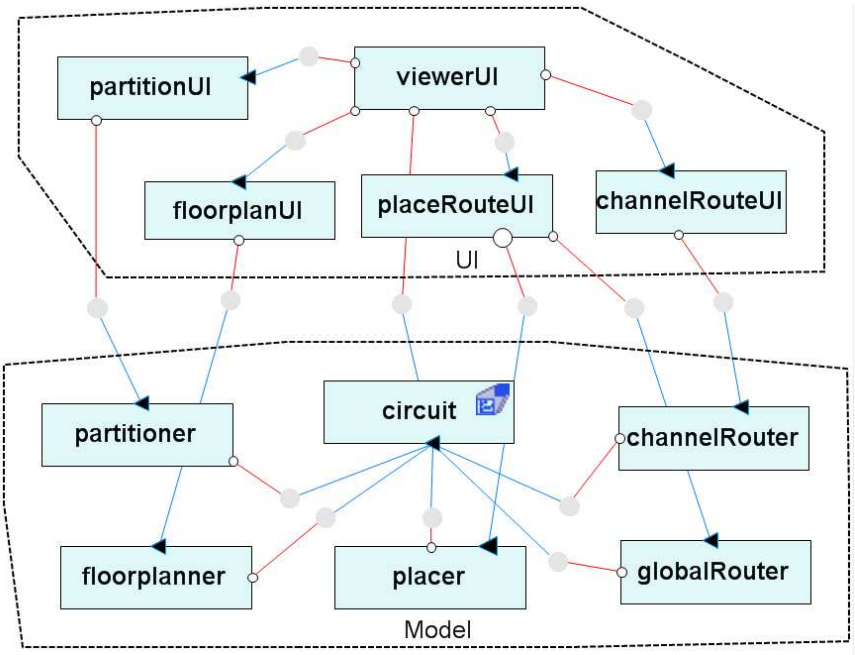


Figure 7.7: Aphyds: designed architecture in Acme.

7.5.1 Modeling the Target Architecture

I documented the Aphyds designed architecture in the Acme ADL based on an informal diagram by the original Aphyds Java developer (Fig. 1.1, Page 3), but iterated the process a few times. When connecting two components in a group, I initially forgot to put the connector into that group[2], which resulted in badly matched connectors.

In an early iteration, I set the analysis to add the divergent components to the conformance view, and noticed a partitionUI divergent component. For consistency, since floorPlanUI and placeRouteUI interact with floorplanner and placeRouter, respectively, I added to the designed architecture a partitionUI that interacts with partitioner, even though the informal drawing omitted partitionUI. The resulting designed architecture is in Fig. 7.7.

7.5.2 Iteration 1

7.5.2.1 Adding Annotations

I initially organized the Aphyds objects into two top-level domains. `UI` holds a `ViewerUI` object and several subsidiary user interface objects. `MODEL` holds a `Circuit` object and computational objects that act on it, such as `Floorplanner`.

I also defined several private domains to hold objects encapsulated by their parent, such as `Map` objects inside a `Circuit` object (Fig. 7.8). These annotations produce a hierarchical OOG, as the (+) sign indicates (Fig. 7.9), but one that still has many objects in the top-level domains.

[2] A predicate in Acme can enforce this rule, and generate a warning when this happens.

```
1  class Circuit<OWNER> { // Implicit parameter
2  // MODEL Circuit c; ⟹ OWNER = MODEL
3    domain OWNED; // Private domain
4    OWNER Node node; // Make peer to self
5    // Declare reference to Map object in OWNED
6    // Inner OWNER annotation is for map elements
7    // String objects have manifest ownership
8    OWNED Map<String,OWNER Node> nodes;
9  }
10 class Circuit {
11   public domain DB; // Public domain
12   domain OWNED; // Private domain
13   DB Node node;
14   OWNED Map<String,DB Node> nodes;
15 }
16 class Node<OWNER> {// Implicit parameter
17   domain OWNED; // Private domain
18   OWNED Vector<OWNER Terminal> terms;
19 }
20 class Net<OWNER> {// Implicit parameter
21   domain OWNED; // Private domain
22   OWNED Vector<OWNER Terminal> terms;
23 }
24 class ViewerUI<M> { // Domain parameter
25   M Circuit circuit;
26 }
27 class Main { // Root class
28   domain MODEL, UI; // Top−level domains
29   MODEL Circuit circuit;
30   UI ViewerUI<MODEL> viewerUI;
31 }
```

Figure 7.8: Aphyds: initial annotations during Iteration 1.

7.5.2.2 Extracting Object Graphs

The extracted object graph for Aphyds is in Fig. 7.9. Aphyds did not require using abstraction by types due to its fairly simple inheritance hierarchy.

7.5.2.3 Abstracting into Built Architecture

I used the default options for abstracting an OOG into the built C&C view.

7.5.2.4 Comparing the Built and Designed Architectures

I used the default options for comparing the built and the designed architectures.

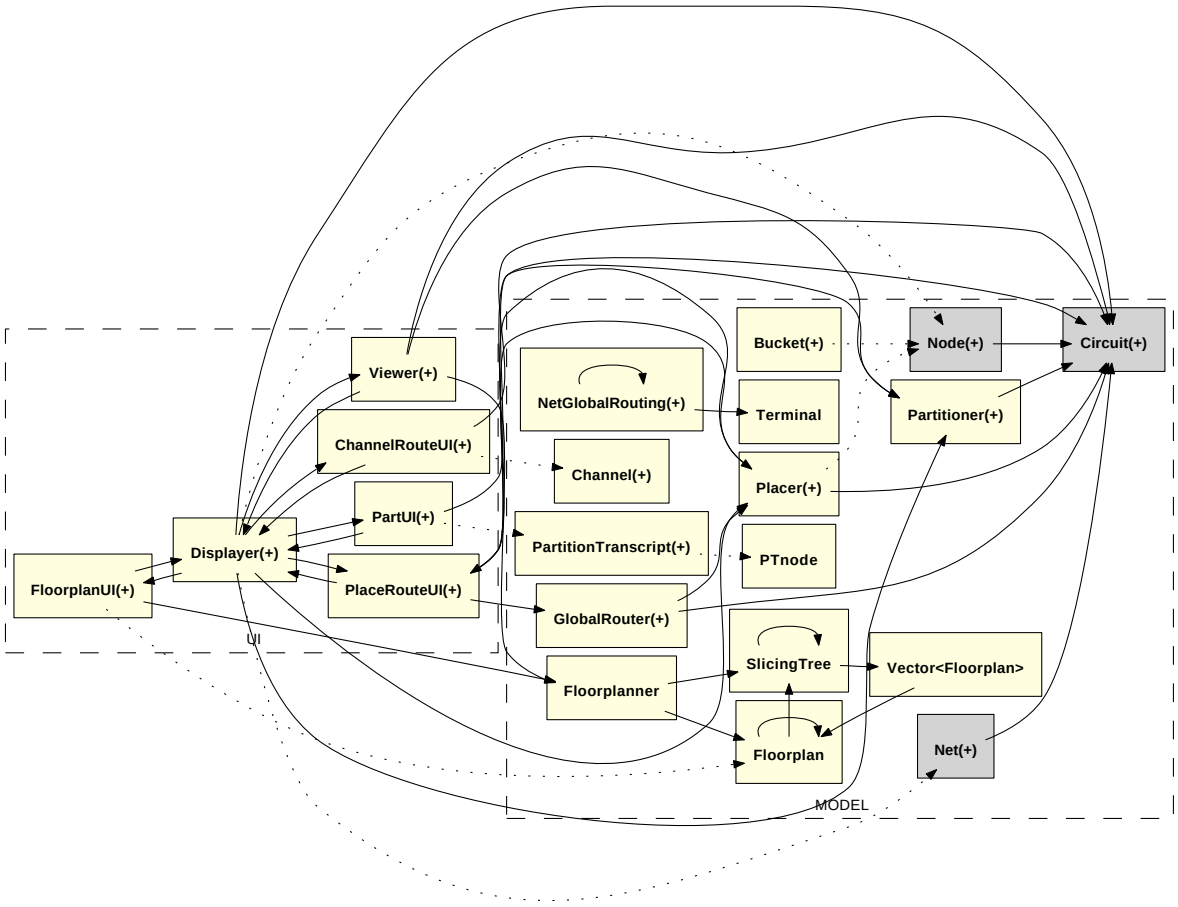


Figure 7.9: Aphyds: OOG using private domains and many peer objects.

7.5.2.5 Analyzing Conformance

The conformance analysis produces neither a readable conformance view (Fig. 7.10) nor good conformance metrics (Table 7.1). For example, `Node` and `Net` are peers of `Circuit` instead of being in its substructure (Fig. 7.8). So the analysis marks as absent the node and net components inside circuit, hence the 2 node absences.

The built view has many more components in the top-level tiers than the designed view, which explains the high node divergence (DN is 11). Moreover, the conformance analysis generates many summary connectors (SE is 97) to account for possible transitive communication, which leads to a high number of edge divergences (DE is 89).

For example, `Displayer` communicates with `Terminal`, and `Terminal` with `Placer`. In reality, `Terminal` is part of `Circuit`, and `Circuit` already communicates with `Placer`. Ideally, the analysis should just mark as convergences the connection between `Displayer` and `Circuit`, and the one between `Circuit` and `Placer`. Since the analysis lacks information about logical containment, it shows instead a divergent summary connector from `Displayer` to `Placer`, and many others. This turns the conformance view into an unreadable fully-connected graph. The low CCM and the 97 summary edges (SE) may not necessarily mean that the designed view is only 21% accurate, but that the built architecture is not yet meaningfully comparable to the

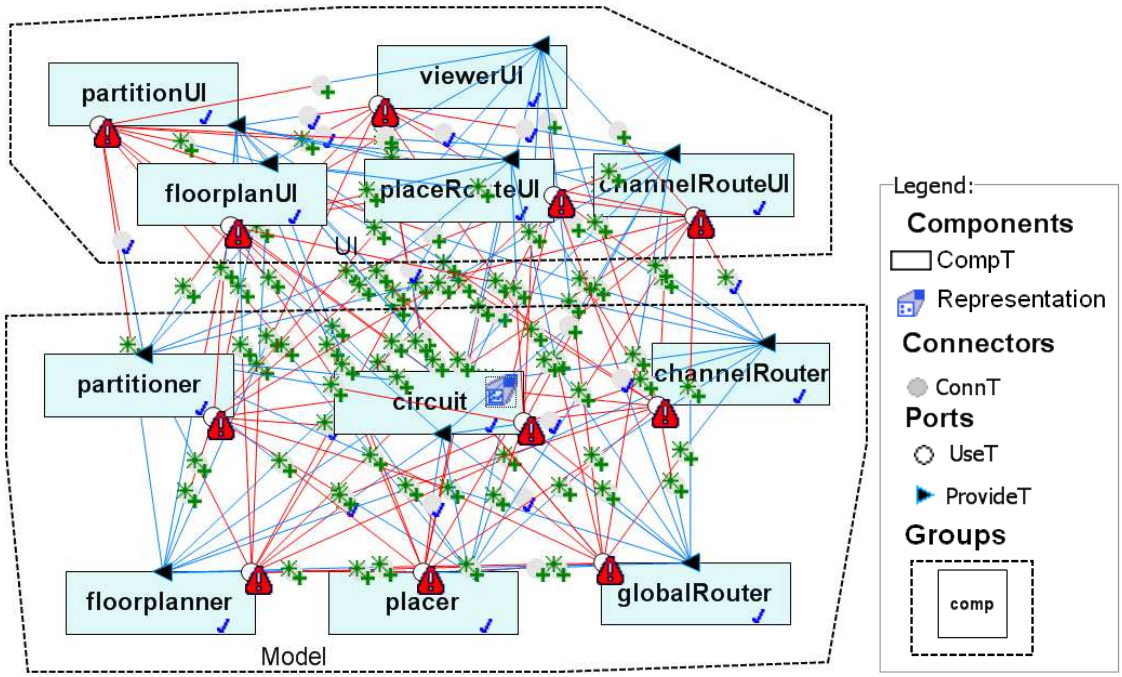


Figure 7.10: Aphyds: conformance view during Iteration 2.

designed one.

7.5.3 Iteration 2

In SCHOLIA, a developer controls the architectural abstraction using annotations. So during Iteration 2, I refined the annotations to get a better match without changing the code.

7.5.3.1 Adding Annotations

Using the designed architecture as a guide (Fig. 1.1), I defined several public domains. Some public domains contain objects that should not be in the top-level tiers. For example, `Viewer` has a `DISPLAY` public domain to hold a `Displayer` object. `Displayer` is not in the developer's diagram (Fig. 1.1), but is not encapsulated either. `Displayer` is only logically contained inside `ViewerUI`, as many other `UI` objects, such as `FloorPlanUI`, reference it directly.

Other public domains abstract low-level objects into more architecturally relevant ones. For example, `Circuit` holds objects such as `Node` and `Net` inside its `DB` public domain, to reflect the designed architecture (Fig. 7.7).

I also created public domains (not shown) as follows:

- `CircuitViewer.DISPLAY`: a public domain on `CircuitViewer` to hold a `Displayer` object that all the other objects in the `UI` domain had references to;
- `Partitioner.DATABASE`: a public domain on `Partitioner` to hold `PartitionTranscript` and `PTnode` objects;
- `Floorplanner.DATABASE`: a public domain on `Floorplanner` for objects such as `SlicingTree`;

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

Figure 7.11: Aphyds: refined annotations during Iteration 2.

- `GlobalRouter.DATABASE`: a public domain on `GlobalRouter` to hold `NetGlobalRouting` objects.

I also reduced the clutter by pushing low-level objects such as `Vector<Floorplan>` into private domains or by passing them linearly between objects.

In most cases, defining a public domain required mostly local and incremental changes to the annotations. With the refined annotations, many objects that were in the `MODEL` top-level domain, such as `Node`, `Net` and `Terminal`, moved into public domains of other objects, such as `Circuit` (Fig. 7.12). As a result, both the extracted OOG and the abstracted built view now have a system decomposition that is closer to the desired architecture (Fig. 7.7). The reader can visually compare the annotations I used in Iteration 1 (Fig. 7.8) to those I adopted in Iteration 2 (Fig. 7.11) and confirm that the changes are fairly local.

Expressiveness challenges. For Aphyds, WARN% is 5%. The remaining annotation warnings are due to expressiveness challenges in the ownership domain type system, similar to those I discussed in Section 4.6.1.3 (Page 135). We believe these warnings do not contribute to missed architectural violations. A warning potentially corresponds to a missed architectural violation, if fixing the warning could produce an additional edge in the extracted object graph, or an additional divergence in the conformance view.

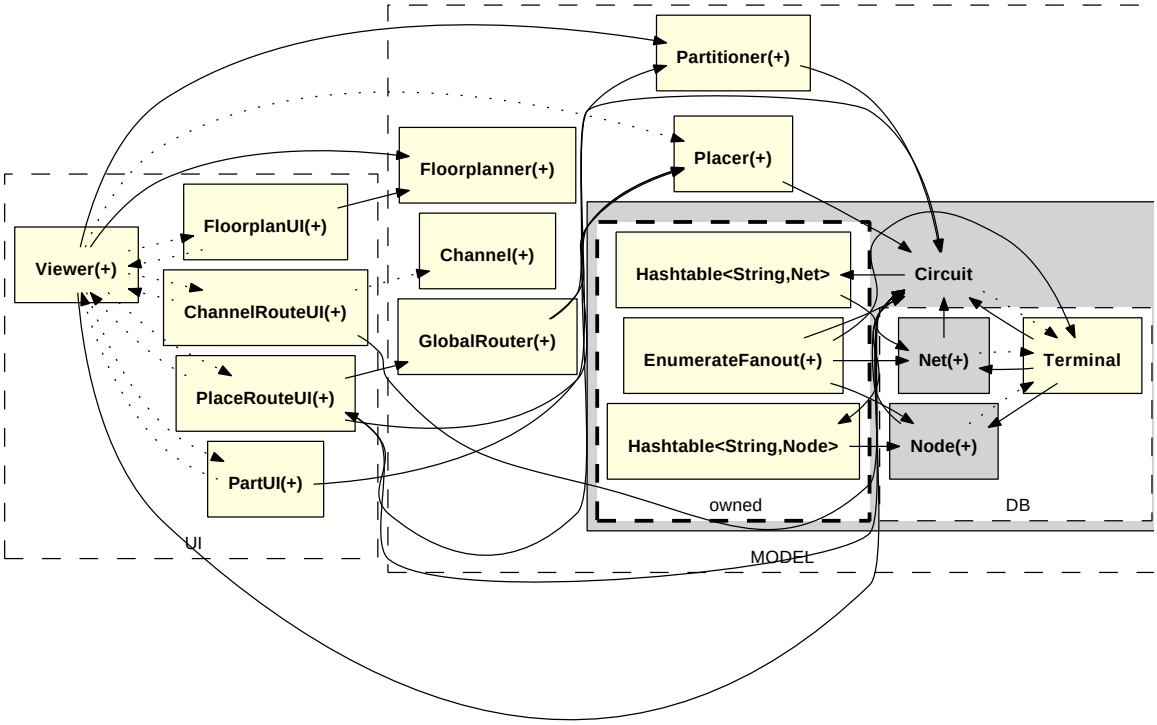


Figure 7.12: Aphyds: refined OOG after defining public domains.

7.5.3.2 Extracting Object Graphs

The extracted object graph, based on the refined annotations, is in Fig. 7.12.

7.5.3.3 Abstracting into Built Architecture

I used the default options for abstracting an OOG into the built C&C view.

7.5.3.4 Comparing the Built and Designed Architectures

I used the default options for comparing the built and the designed architectures. However, the structural comparison did not correctly match `Node` and `Net` in the built view, to node and net in the designed view (Fig. 6.7, Page 213). Indeed, this is a case where the graph structure not rich enough to give a good structural match. This was fixed by manually forcing `Node` to match node, and `Net` to match net, respectively.

7.5.3.5 Analyzing Conformance

In Iteration 2, the conformance analysis matched the components better, with 0 node absences and 1 node divergence, which corresponds to `Terminal`. The analysis now marks as convergent, both node and net inside circuit, as well as the connectors between them (Fig. 7.13). In the built system, node and net do not communicate directly, but do so through `Terminal`. So the two convergent connectors inside circuit have the summary decoration ✳. As an aside, the edges

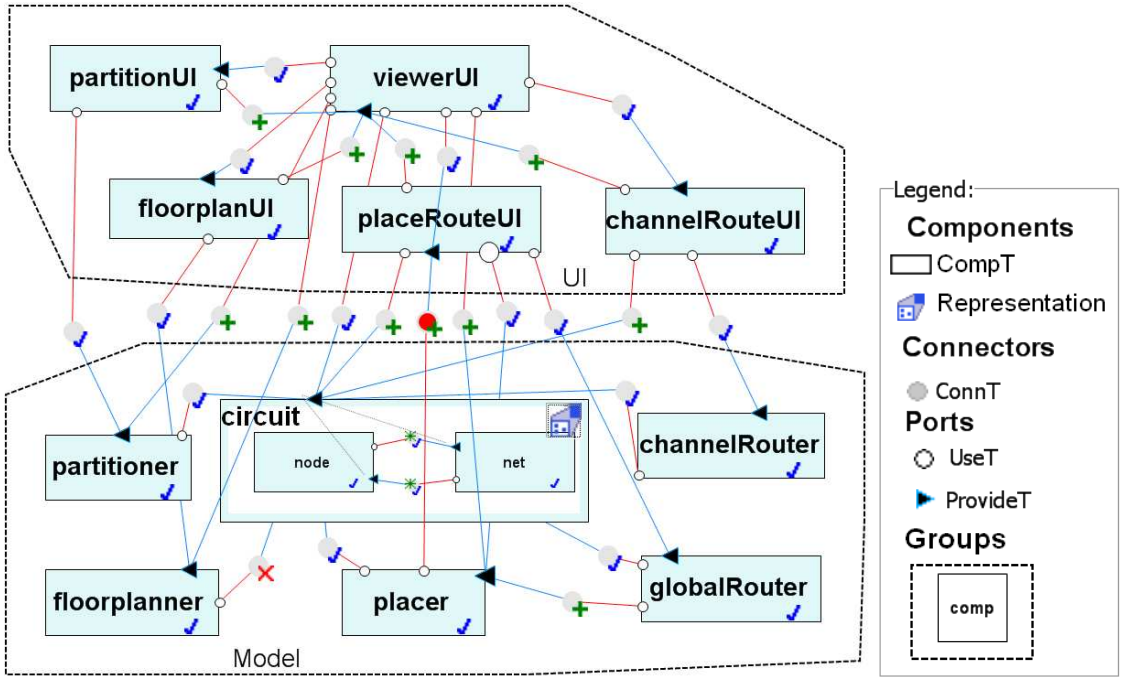


Figure 7.13: Aphyds: conformance view during Iteration 2.

from `Node` to `Terminal` and from `Net` to `Terminal` are in fact lifted edges. This example justifies the different kinds of edge summarization: lifting an edge in an OOG and in a C&C view, then adding summary connectors in the C&C view. In fact, we adapted the earlier explanation of summary connectors (Fig. 6.7, Page 213) from this part of Aphyds.

7.5.4 Summary of Findings

As one would expect from an informal diagram, the designed architecture (Fig. 7.7) is only about 60% accurate, based on the CCM metric. Indeed, SCHOLIA identified a divergent component partitionUI, several divergences between viewerUI and other UI components, between UI and MODEL components, and between MODEL components. Many connections which the developer thought to be uni-directional were bi-directional in reality. A developer could use ArchMod to add the divergent connectors to the designed architecture.

One divergence that crosses tiers, from placer in `MODEL` to placeRouteUI in `UI`, was a red flag and a potential concurrency bug (this is the connector I manually set to be darker in shade in Fig. 7.13). As a multi-threaded application, Aphyds must respect certain framework-specific conventions to call back from a worker thread executing a long-running operation into the user interface thread. I used CodeTraceJ to trace this divergence to a `PlaceRouteUI` field inside class `Placer`, and manually inspected that the code handled the callback correctly.

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

Table 7.1: Aphyds conformance metrics. We count convergent nodes (CN), divergent nodes (DN), absent nodes (AN), convergent edges (CE), divergent edges (DE), absent edges (AE) and summary edges (SE). CCM is the core conformance metric.

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

(Iteration 1) and 33 seconds (Iteration 2).

7.5.5 Aphyds Discussion

Threats to experimental validity are classified as *internal*, whether the results were determined by the technique or by some other factor, or *external*, to what extent the results can be generalized. In this section, I mainly discuss internal validity, and defer the discussion of external validity until after I present the other evaluations (Section 7.9.1).

To what extent are the results of this case study due to our knowledge of the Aphyds code, and to what extent are they due to using SCHOLIA? Although I did not author the Aphyds system, I previously read about re-engineering Aphyds to ArchJava (Aldrich et al. 2002a) and studied its ArchJava version (Abi-Antoun et al. 2008). But I believe the results of this case study are due to using SCHOLIA and not to any previous knowledge of the code. The 8,000-line code base is too non-trivial for anyone to hold in his head at once. Moreover, when I studied Aphyds previously, I represented the desired architecture differently (Abi-Antoun et al. 2008, Fig. 19). I did not express tiers, had one `model` component with planner, partitioner and others as sub-components, and did not represent circuit's substructure[3].

Although the experimenter also designed several of the tools, a typechecker kept him honest. He could not insert an arbitrary annotation without getting a warning, or otherwise manipulate the extracted architectures.

Another threat is that an electrical engineering professor—rather than a professional architect—drew the Aphyds intended architecture. However, we mined the diagram only for which objects are architecturally significant, the top-level tiers it shows, and the hierarchical system decomposition it uses for `Circuit`'s substructure, all general concepts in modeling architectures (Clements et al. 2003).

Another confound is whether the built and the designed architectures represent the same information. For instance, when we redrew the original developer's diagram (Fig. 1.1), we reversed the direction of some arrows (Aldrich et al. 2002a, p. 192) and excluded data flow edges. For a meaningful conformance analysis, the designed and the built architectures must have the same kind of connectors, here, points-to relations.

[3]I did not previously have node and net inside circuit due to ArchJava's following limitation: in the ArchJava code, different components share instances of `Net` and `Node`. Thus, neither `Net` nor `Node` is an ArchJava `component class`. As a result, the tool that extracts a C&C view from ArchJava code does not show net or node sub-components inside circuit because the tool shows only the instances of ArchJava `component classes`.

Can SCHOLIA identify at least as many violations as the state-of-the-art in the *static* enforcement of *runtime* architectures? The state-of-the-art would be library-based (Medvidovic et al. 1996) or language-based (Aldrich et al. 2002b; Schäfer et al. 2008) solutions. For instance, the C2 ADL mandates a specific architectural framework (Medvidovic et al. 1996), but requires developers to follow strict guidelines to avoid introducing architectural violations. There are no tools to check that an implementation obeys those rules (N. Medvidovic, personal communication, 2008). Language-based solutions, first exemplified by ArchJava, radically extend the language to incorporate architectural components and ports, and enforce communication integrity using a type system (Aldrich et al. 2002b; Schäfer et al. 2008).

Using SCHOLIA, we found all the violations that (Aldrich et al. 2002a) previously found for the same system. However, (Aldrich et al. 2002a) found the architectural violations in Aphyds, only after they re-engineered the implementation to ArchJava.

Many factors make re-engineering a typical Java implementation to ArchJava hard (Aldrich et al. 2002a; Abi-Antoun and Coelho 2005; Abi-Antoun et al. 2007a). In ArchJava, a developer makes an object architecturally significant by making its declared type a `component class`. But ArchJava prohibits taking a reference to any instance of a `component class` as an argument, or returning a reference to one. Also, the developer may define additional ArchJava `component classes`, just to capture the intended system decomposition. In a C&C view extracted from ArchJava, Component a appears inside Component b if a instantiates b as one of its fields. As a result, one may define additional `component classes` just to capture the intended system decomposition. Finally, an architecture extracted from an ArchJava implementation shows only instances of component classes and the architectural system decomposition they prescribe. A developer cannot drill down into each component until she reaches leaf objects that are typically data structures.

Deciding ahead of time which objects are architectural components and which objects are data structures and should be left as regular Java `classes`, achieving the desired decomposition, and respecting ArchJava's restrictions make re-engineering to ArchJava harder than simply converting each Java `class` into an ArchJava `component class`.

Indeed, Aldrich identified as an area of future work for the ArchJava project, the need to address "the dichotomy between the component world and the object world—two different kinds of entities with different rules" (Aldrich 2003). In ArchJava, classes that have many instances that are shared or passed between different instances are best left as ordinary Java classes, because ArchJava's `component classes` may be too restrictive in those cases.

SCHOLIA does not have instances of `component classes` that are distinguished from instances of regular Java `classes`. As a result, SCHOLIA does not have the dichotomy that exists in ArchJava, since SCHOLIA achieves hierarchy using annotations and without additional classes. In SCHOLIA, all objects are instances of regular Java classes, and there are fewer restrictions on passing object references. In SCHOLIA, an object becomes secondary to another object by being inside one of the domains of that object. The more architectural objects are higher in the ownership hierarchy. In particular, logical containment can impose an arbitrary hierarchy on an object graph, and allows SCHOLIA to support arbitrary object-oriented code better.

To apply SCHOLIA to Aphyds, we only added annotations to the code. In contrast, for Aphyds, Aldrich et al. specified within the code over 20 ArchJava `component classes` and over 80 `ports`, re-engineered the program to obey the type system's restrictions, and inadver-

tently injected defects.

Could any other *static* conformance analysis approach find the violations that SCHOLIA found? It is a genuine threat to validity to compare a designed runtime architecture to a built code architecture, or vice versa. All previous *static* conformance approaches deal with the *code architecture* (Feijs et al. 1998; Laguë et al. 1998; Murphy et al. 2001; Sangal et al. 2005; Eichberg et al. 2008). The closest to a *statically* extracted *runtime architecture* for an object-oriented system would be an object graph extracted by a static analysis, whether it uses annotations (Lam and Rinard 2003) or not (Jackson and Waingold 2001; O'Callahan 2001). Most flat object graphs would not convey sufficient architectural abstraction to be used for conformance analysis. Of course, we could compare SCHOLIA's results to those obtained by a *dynamic* analysis (Sefika et al. 1996b; Schmerl et al. 2006). But a dynamic analysis cannot claim to represent all possible executions.

Could a static conformance approach for the *code architecture* detect all the violations in a *runtime architecture*? For example, *could Reflexion Models (RM) (Murphy et al. 2001; JRM 2003) find all the violations that* SCHOLIA *found?* In fact, we modeled SCHOLIA closely after RM, a standard in analyzing the conformance of code architectures. In RM, a third-party tool extracts a *source model* from the implementation. A developer posits a designed *high-level model* and a *map* between the source and high-level models. RM pushes each interaction described in the source model through the map to infer edges between high-level model entities. RM then compares the inferred edges with the edges stated in the high-level model. There are many similarities between SCHOLIA and RM. For example, WARN is similar to RM's tracking of unmapped entries in the source model. The major difference is that RM is designed for the code architecture. There are also several minor differences. For example, RM has no divergent or absent nodes. In RM, if the map generates a node that is not the designed view, RM automatically adds that node to the designed view. In other words, RM has no divergent or absent nodes, nor does it compute summary edges.

In Aphyds, many important classes are instantiated once, so the object graph is somewhat similar to a class diagram with associations. Of course, there are still non-trivial differences related to the different instantiations of the various container classes such as `Vector`. Out of curiosity, we ran JRM (JRM 2003) on Aphyds. JRM supports neither tiers nor hierarchical target architectures, so we used a simplified high-level model that did not include tiers and ignored `Circuit`'s substructure. In some cases, JRM's finding was consistent with what SCHOLIA found. For example, RM found the divergence from placer to placeRouteUI, because it corresponds to a direct field reference declared in class `Placer`.

However, JRM produced many divergences and absences, and many were false positives and false negatives, because it does not show an edge between two high-level elements if they communicate through a chain of objects. For example, a `ViewerUI` object does not directly point to a `FloorPlanUI` object. Instead, a `ViewerUI` points to a `Displayer`, and `Displayer` references a `FloorPlanUI`. Moreover, `Displayer` is in a public domain of `ViewerUI`. When `ViewerUI`'s substructure is elided, the OOG *lifts* that relation to `ViewerUI`, and shows a *lifted edge* from `ViewerUI` to `FloorPlanUI`, shown as a dotted edge in the OOG (Figs. 7.9, 7.12).

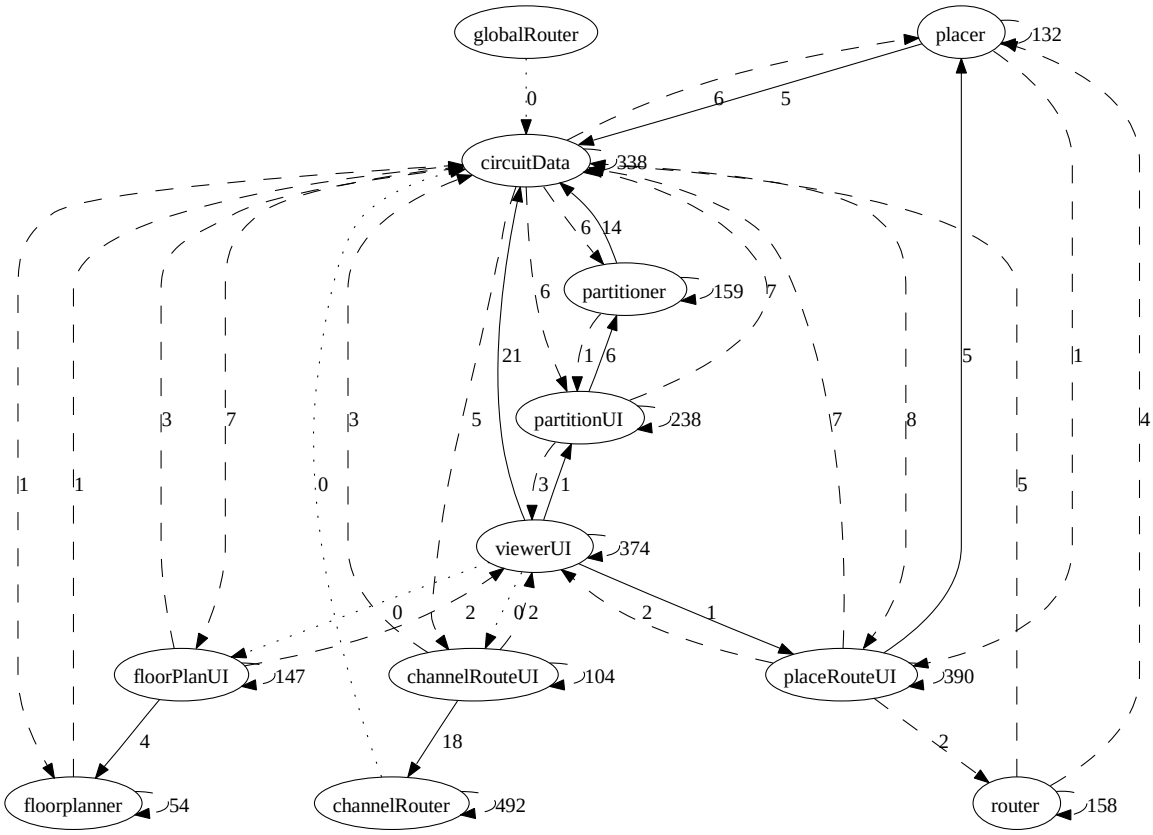


Figure 7.14: Aphyds: results using the Reflexion Models tool (JRM 2003).

However, RM showed *absences* between viewerUI and floorPlanUI (See Fig. 7.14)—instead of the correct divergences and convergences.

Similarly, RM would not correctly handle circuit's substructure. Unlike RM, SCHOLIA distinguishes the `Vector` of `Terminals` inside `Net`, from the one inside `Node`, and uses object pulling, object merging, edge lifting, and edge summarization to check the communication between node and net. In general, a tool for the code architecture cannot handle the runtime architecture.

Does SCHOLIA generate many false positives? False positives are possible in general, as with any sound static analysis, but SCHOLIA attempts to reduce them. For example, the edges in an OOG are more precise than super-imposing associations from a class diagram. Also, SCHOLIA checks only matching substructures, and not the entire object hierarchy. There are several sources of false positives in SCHOLIA. The OOG extraction uses a whole-program and not a reachability analysis that excludes infeasible paths. Also, the conformance analysis may add summary edges that are false positives, as in Iteration 1 which had 97 summary edges. But if the built and the designed architectures have a similar hierarchical decomposition and a similar number of components at each hierarchy level, the analysis adds fewer summary edges. Indeed, Iteration 2 had only 2 summary edges, and neither one was a false positive. In our Aphyds evaluation, we used CodeTraceJ to trace each finding to the code, and confirmed that it does not correspond

to an obvious false positive. Aphyds was written by a professor for one of his classes. So this may explain the absence of infeasible paths. In addition, Aphyds has no interesting inheritance hierarchy. As a result, the imprecision due a field assignment through a superclass (Section 2.6.3) and the lack of object-sensitivity in the OOG extraction do not show up in this case study.

Furthermore, to account for false positives, techniques such as Reflexion Models typically support manual input (Murphy 1996, pp. 84–88). When studying the conformance findings, a developer can manually override any finding and specify a reason for the override. For each manual override, a tool can store in the designed architecture the original finding, the overridden finding, and the reason for the override, together with any associated traceability information. When re-running the analysis, if a computed edge has a manual override associated with it, the analysis can compare the traceability of the computed edge to the previously saved one, and raise a warning if it detects a discrepancy. Similarly, our tools could support such features to annotate or override a conformance finding (Abi-Antoun et al. 2006).

7.6 Extended Example: JHotDraw

We now analyze the conformance of the JHotDraw subject system we previously discussed in (Section 4.6, Page 128).

7.6.1 Modeling the Target Architecture

As is the case for many legacy systems, we were unable to find a documented runtime architecture for JHotDraw. We did find however a documented abstracted code architecture (Fig. 4.2). Of course, the runtime architecture may be significantly different from the code architecture. We used the code architecture as an estimate to be refined by the conformance analysis step.

For each class in the code architecture, we created a component instance. Then, for each association in the class diagram, we created a connection between the corresponding components (Fig. 7.15).

JHotDraw's architecture posed another challenge. From a previous study, we knew that a `Drawing` was actually implemented as a `Figure`, contrary to the designed code architecture. So the OOG, and thus by transformation, the built view, represented both `Drawing` and `Figure` with one runtime component. Had we modeled `Drawing` and `Figure` as separate in the designed view, the structural comparison would not have detected the splitting or merging (Chapter 5). This led us to merge `Drawing` and `Figure` into one `DrawingFigure` component in the designed view.

7.6.2 Adding Annotations

We discussed the JHotDraw annotation process in Section 4.6 (Page 128).

7.6.3 Extracting Object Graphs

I discussed various JHotDraw object graphs in Section 4.6 (Page 128). For the conformance analysis, I chose an object graph with abstraction by design intent types (Fig. 4.24).

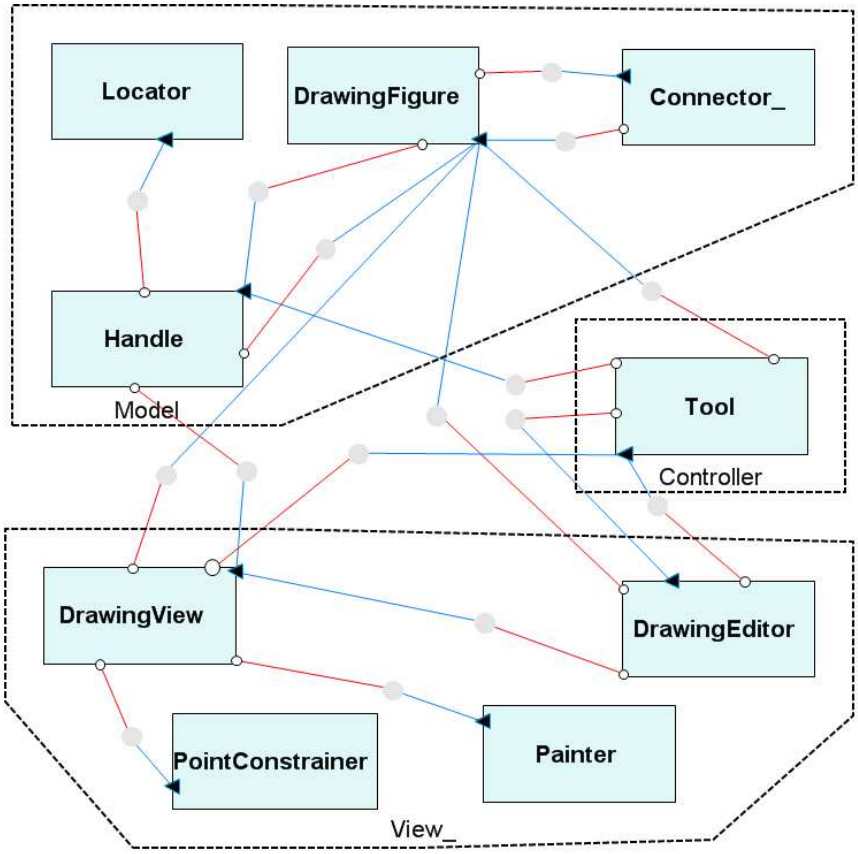


Figure 7.15: JHotDraw: designed architecture documented in Acme.

7.6.4 Abstracting into Built Architecture

I used the default settings. The result is in Fig. 7.16.

7.6.5 Analyzing Conformance

ArchConf detected many renames and a few missing components, such as an `Undoable` in `CONTROLLER` and an `UndoManager` in `MODEL`. Many connections, we thought to be unidirectional, such as between components `DrawingView` and `DrawingEditor`, turned out to be bi-directional (Fig. 7.17).

When the built and the designed architectures do not have roughly the same number of top-level components and architectural decomposition, the summary connectors can make the graph unreadable. For this reason, ArchConf provides the option of turning off the generation of summary connectors. Of course, in that case, the conformance view no longer guarantees communication integrity. Turning off summary connector might still be useful as an intermediate step, if a developer cares only about the core objects in the designed architecture (Fig. 7.18), or while she is refining the target architecture.

There was however one big surprise: there were no callbacks from `MODEL` into `CONTROLLER`! In the base MVC pattern, a controller registers itself with the model and receives notifications.

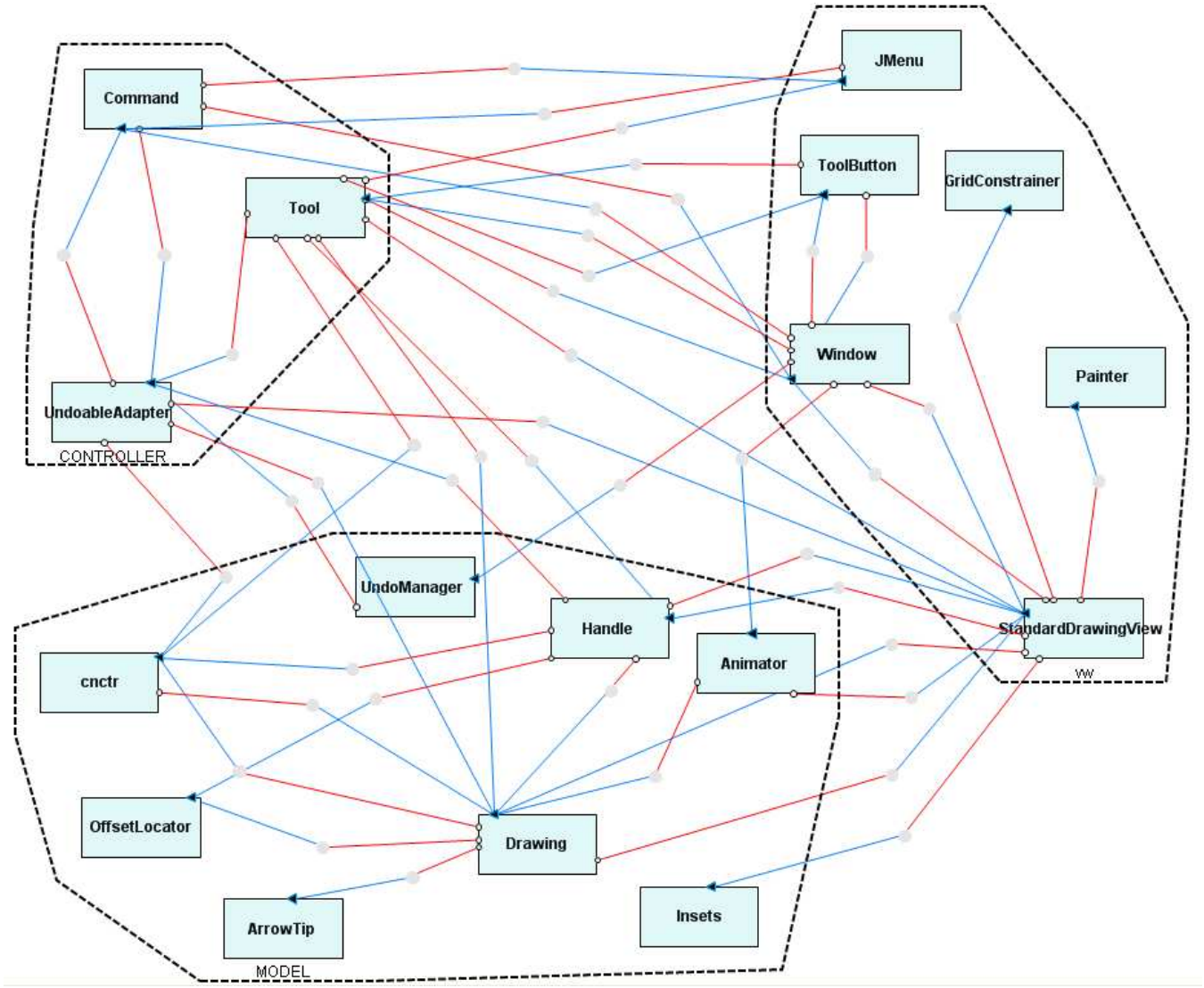


Figure 7.16: JHotDraw: built architecture in Acme.

Since there is no controller component, we suspected that the view acts also as controller, a common implementation optimization. Indeed, in the JHotDraw "CRC Cards View", the designers mention that `DrawingView` "handles input events" (Gamma 1998, Slide #10), a controller responsibility.

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

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

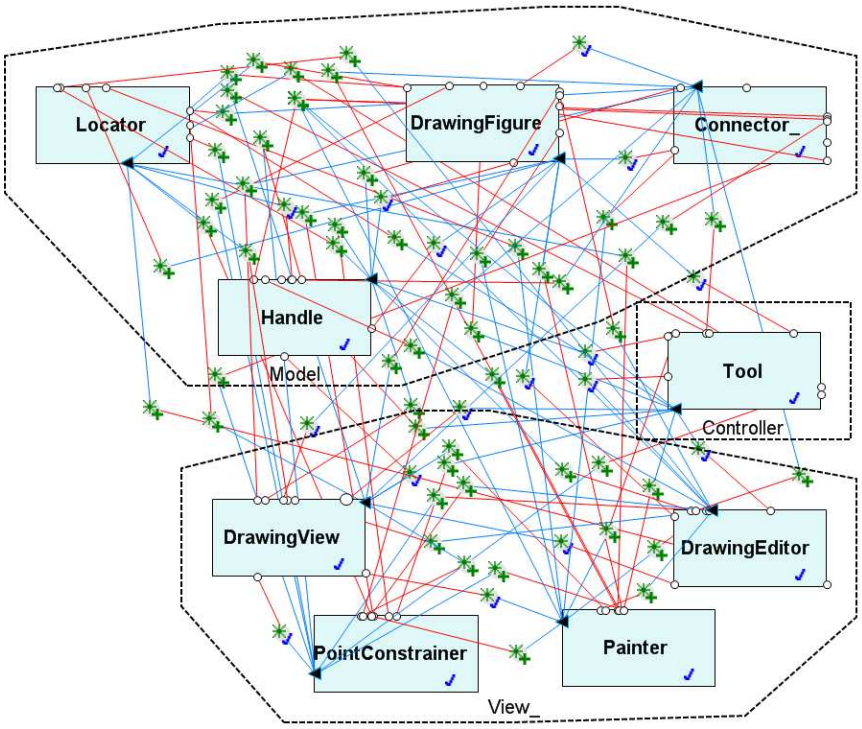


Figure 7.17: JHotDraw: conformance view with summary edges.

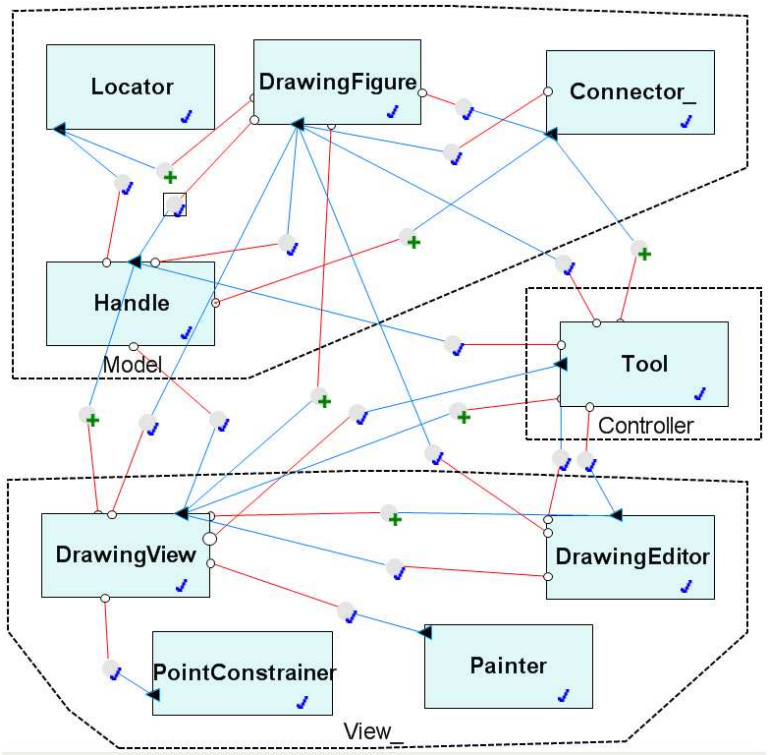


Figure 7.18: JHotDraw: conformance view without summary edges.

Table 7.2: JHotDraw conformance metrics.

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

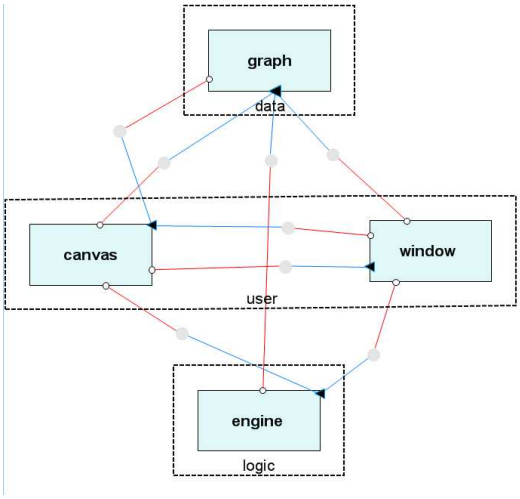


Figure 7.19: HillClimber: designed architecture.

7.6.6 Summary of Findings

Metrics. The low CCM indicates a large proportion of divergences and absences (Table 7.2). This was expected because of how we obtained the designed view. Moreover, the designed view is missing several top-level components, in each of the tiers.

7.7 Extended Example: HillClimber

We now analyze the conformance of the HillClimber subject system we previously discussed in (Section 4.7, Page 154).

7.7.1 Modeling the Target Architecture

I based the designed HillClimber architecture on available documentation (Fig. 7.19).

In HillClimber, the application *window* uses a *canvas* to display *nodes* and *edges* of a *graph* to show the output of a computational *engine*. Based on a hint from one of the original framework developers, we posited in the target architecture that the `engine` component need not connect to `window` or `canvas`.

7.7.2 Adding Annotations

I previously discussed the HillClimber annotation process in Section 4.7 (Page 154).

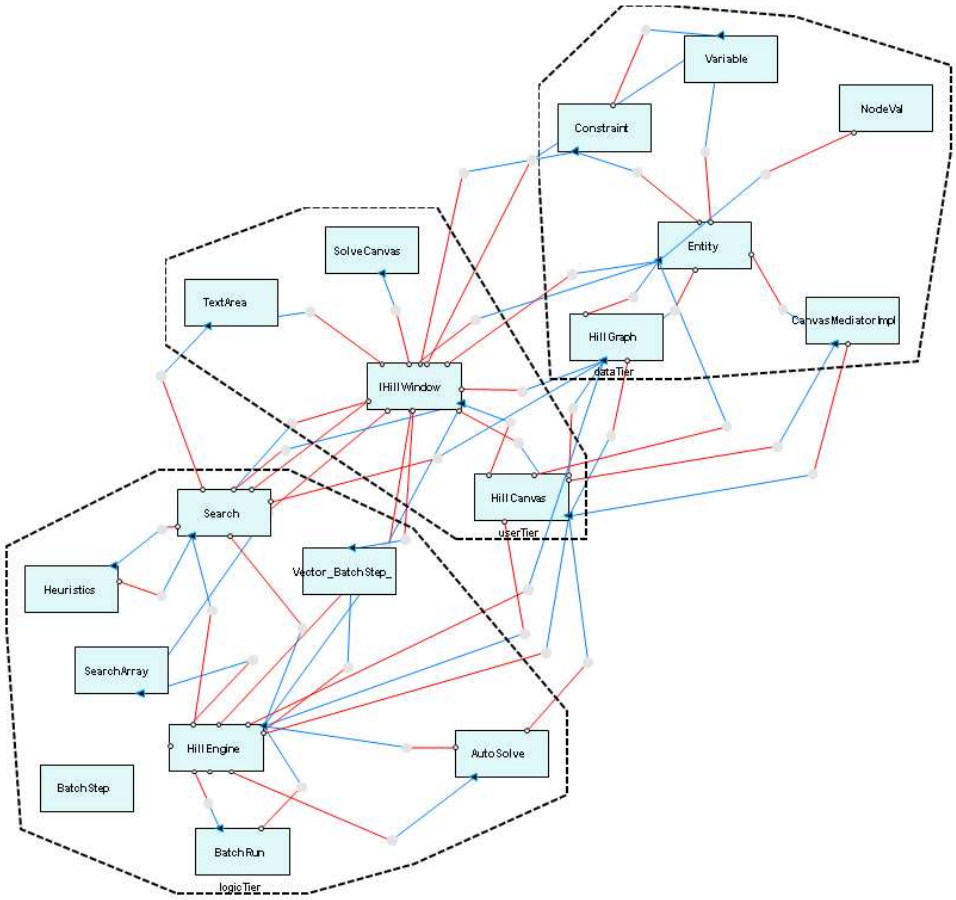


Figure 7.20: HillClimber: built architecture in Acme.

7.7.3 Extracting Object Graphs

I discussed extracting object graphs from HillClimber in Section 4.7 (Page 154).

7.7.4 Abstracting into Built Architecture

I used the default settings. The result is in Fig. 7.20.

7.7.5 Analyzing Conformance

The conformance analysis confirms that `engine` connects to both `window` and `canvas`, contrary to the designed architecture (Fig. 7.21).

7.7.6 Summary of Findings

Metrics. The CCM is high since very few edges were affected (Table 7.3). The high node divergence is due to a designed view that has fewer elements at the top-level than the built view.

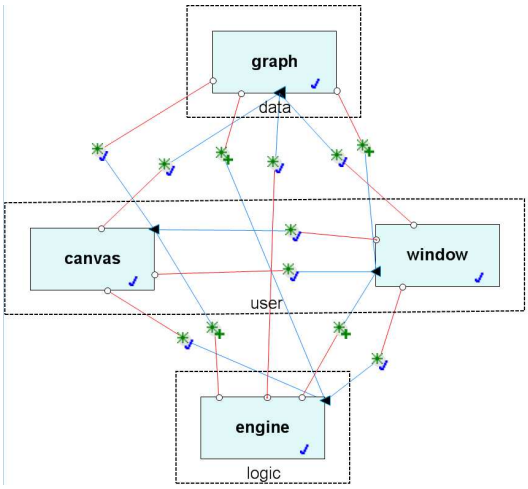


Figure 7.21: HillClimber: conformance view.

Table 7.3: HillClimber conformance metrics.

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

The developer must either enrich the designed view by representing additional components in the `LOGIC` tier, or refine the annotations to push more components in the `LOGIC` tier into `engine`'s substructure.

7.8 Extended Example: CryptoDB[4]

This case study is an application of SCHOLIA to analyze conformance between a Java implementation and a security runtime architecture, entirely statically and using annotations. We also illustrate enforcing constraints both at the code level and architecturally. The subject system is CryptoDB, a secure database system designed by a security expert (Kenan 2006). CryptoDB follows a database architecture that provides cryptographic protections against unauthorized access, and includes a 3,000-line sample implementation in Java. The presence of both a Java implementation and an informal architectural description make CryptoDB an appropriate choice to demonstrate using SCHOLIA to analyze conformance and enforce structural constraints.

Why this case study? CryptoDB has compelling architectural documentation, and a target architecture designed by a security expert[5]. The target architecture also has richer types, properties and constraints than the previous architectures that I analyzed using SCHOLIA, which increases the external validity of the result. Unlike the previous case studies, I conducted the summative CryptoDB case study to evaluate SCHOLIA after I finished developing the approach. In addition, this case study illustrates the unique strength of SCHOLIA, the ability to analyze statically

[4]Preliminary results of the CryptoDB case study appeared in (Abi-Antoun and Barnes 2009a).

communication integrity, for all possible program runs, which is crucial for the security domain.

Evaluation methodology. During the evaluation, a colleague played the role of the architect, while I played the role of the developer. The architect controlled the target architecture, and the developer controlled the annotations and the code. In particular, the developer was not allowed to change the target architecture directly. Instead, he had to convince the architect that the proposed change was justified architecturally, rather than a workaround to apply SCHOLIA. Also, we forbade ourselves from making changes to the source code, except to annotate it.

7.8.1 Threat Modeling

For many years, companies such as Boeing and Microsoft have been using *threat modeling* (Howard and Lipner 2003; Torr 2005; Howard and Lipner 2006) as a lightweight approach to reason about security, to capture and reuse security expertise and to find security design flaws during development. During threat modeling, development teams construct security architectures that are later reviewed by security experts.

Although threat modeling often finds security design flaws, it suffers from the two problems of architectural extraction and conformance analysis. When a security expert asks a developer to build a security architecture for a system under study, the developer typically produces a diagram mostly from his recollection of how the system works, with little tool support to extract such an architecture from the code. Then, during the security review, the experts study the architecture, assign to the components different architectural properties such as trustLevel (Abi-Antoun et al. 2007b) or privacyLevel, and enumerate all possible communication between the more trusted and the less trusted components of the system. But if the architecture does not show all the communication that is present in the system, the results of an architectural-level analysis may be incorrect. While any architecture-based approach suffers from these problems, security architectures pose special challenges.

A security architecture is an example of a runtime architecture. Moreover, an analysis at the level of a security architecture must consider the worst and not the typical case of possible component communication. Indeed, the analysis results are valid only if the architecture reveals all objects and relations that may exist at runtime, in any program run. This requires a static analysis which can capture all possible executions.

Moreover, SCHOLIA's focus on the communication integrity notion of conformance is also crucial for an architectural-level security analysis. Typically, a security review enumerates all of the possible information flows between trusted and untrusted parts of the system. However, if the analyzed architecture does not satisfy communication integrity, the architecture may not show all information flows that are present in the implementation, and so the architectural analysis cannot be trusted to be correct. Without enforced communication integrity in the target architecture, the source code of the entire system must be painstakingly analyzed, and the architecture provides little benefit for reasoning about the implementation (Aldrich 2003, p. 3).

[5]In contrast, an electrical engineering professor designed the Aphyds target architecture (Aldrich et al. 2002a).

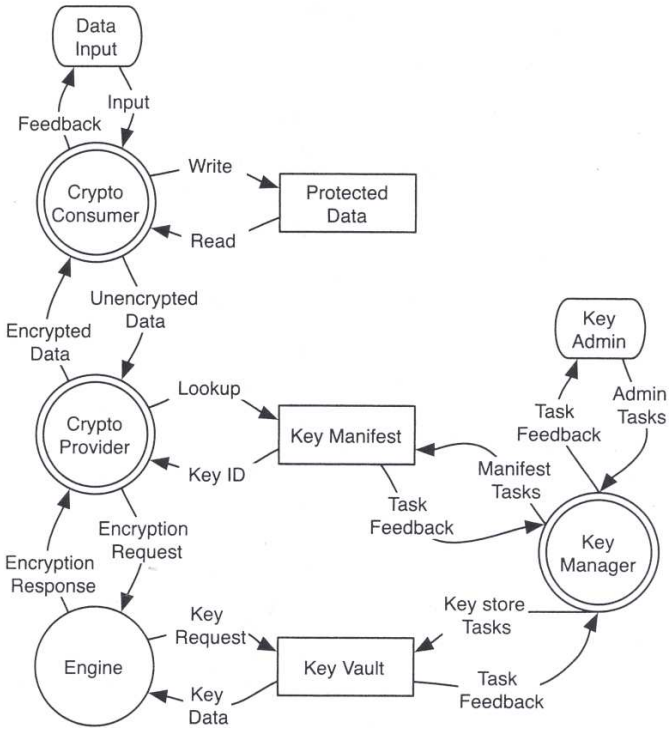


Figure 7.22: CryptoDB: documented Level-1 DFD (Kenan 2006, Fig. 9.1).

7.8.2 Available Documentation

Architectural reasoning about security is best accomplished with a runtime architecture, not a code architecture. A security architecture[6] is an example of a runtime architecture which shows runtime components and connectors, uses hierarchical decomposition, and partitions a system into tiers.

7.8.2.1 Documented Architectures

We studied the architectural documentation available for CryptoDB, which consisted of various Data Flow Diagrams (DFDs) along with accompanying, explanatory text (Kenan 2006). A DFD is a runtime architecture that can be represented as a Component-and-Connector view (Clements et al. 2003, pp. 364–365). Fig. 7.22 is a Level-1 DFD. Fig. 7.23 is a Level-2 DFD which refines in place some of the components from the Level-1 DFD.

We mined the diagrams for the architecturally significant elements, the top-level tiers, and the hierarchical system decomposition. During the course of the study, it also became apparent that the documentation and the code used slightly different terminology. For example, the textbook and DFDs referred to a "key manager", but the code had a `KeyTool`. In the rest of this discussion,

[6]Threat modeling typically uses a Data Flow Diagram (DFD) with security-specific annotations to describe how data enters, leaves and traverses the system by showing data sources and destinations, the processes that data goes through and the trust boundaries in the system (Torr 2005). Here, we use a slightly different architectural style of a security architecture, one which shows points-to (not data flow) connectors, has no explicit data stores or external interactors, and uses more general boundaries that indicate different runtime tiers.

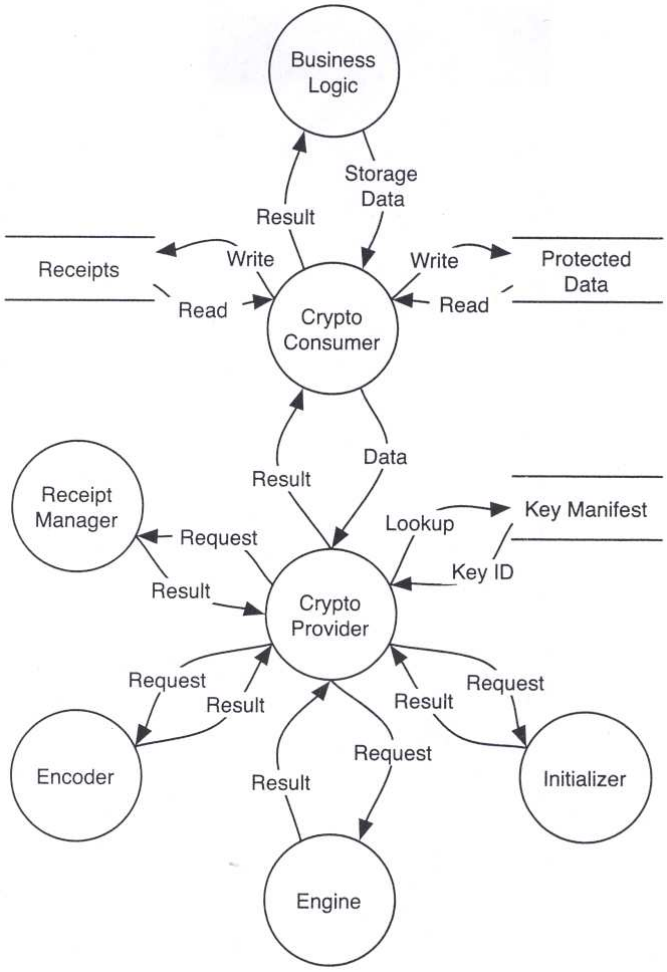


Figure 7.23: CryptoDB: documented Level-2 DFD (Kenan 2006, Fig. 6.1).

we will often use the names from the implementation. Similarly, when we chose the names of the components in the target architecture, we knew that they may not match exactly the names of the code elements, and that SCHOLIA's structural comparison can detect renames. Table 7.4 shows a mapping between the components in the target architecture and the corresponding Java classes. Of course, this mapping is only a first approximation, because one type in the class diagram can map to multiple instances in the architecture; and multiple types in the class diagram can be represented by the same canonical component in the runtime architecture.

7.8.2.2 Code Architecture

I used the Eclipse UML tool (Omondo 2006) to extract from the CryptoDB implementation various views of the code architecture. For instance, Fig. 7.24 shows the CryptoDB package structure. Fig. 7.25 shows a class diagram with a few selected core types from CryptoDB. A quick glance shows that these module views are not very comparable to the security architecture drawn by the system's designer.

Architectural Component	Java Class	Note
CustomerManager	cryptodb.test.CustomerManager	AKA “crypto consumer”
CustomerManager.Receipts	cryptodb.CryptoReceipt	Receipts the consumer holds onto
CustomerInfo	cryptodb.test.CustomerInfo	AKA “protected data”
CryptoProvider	cryptodb.core.Provider	
CryptoProvider.ReceiptManager	cryptodb.CompoundCryptoReceipt	Used by the provider to produce receipts
CryptoProvider.Encoder	cryptodb.Utils	
EngineWrapper	cryptodb.core.EngineWrapper	
EngineWrapper.Engine	javax.crypto.Cipher	
KeyManifest	cryptodb.KeyAlias	The key manifest contains key aliases
KeyVault	cryptodb.core.LocalKeyStore	The key vault contains keys (LocalKeys)
KeyManager	cryptodb.KeyTool	

Table 7.4: CryptoDB: mapping between architectural components and code elements.

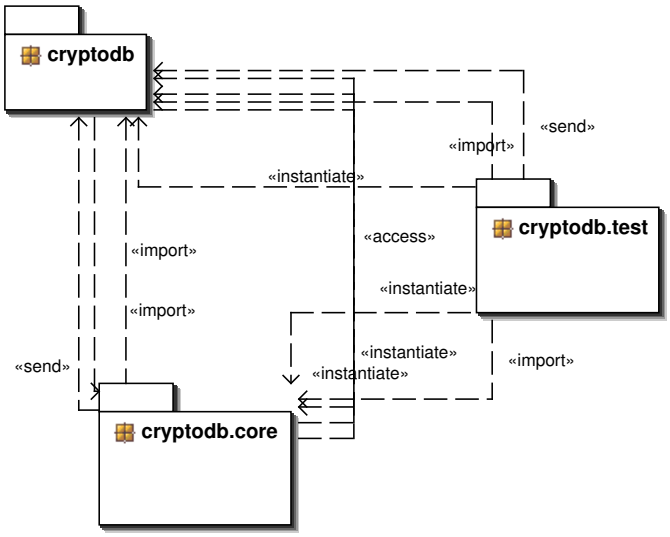


Figure 7.24: CryptoDB: layer diagram.

7.8.2.3 Flat Object Graphs

We also used available tools to extract CryptoDB object graphs. As mentioned earlier, non-hierarchical object graphs mix low-level objects such as `HashMap` with architecturally relevant objects such as `CryptoReceipt`, and a developer has no easy way to distinguish them. These flat object graph are unreadable, even for a small 3,000-line program, and do not convey sufficient architectural abstraction to enable analyzing conformance. I obtained Fig. 7.26 using PANGAEA (Spiegel 2002), and Fig. 7.27 using WOMBLE (Jackson and Waingold 2001).

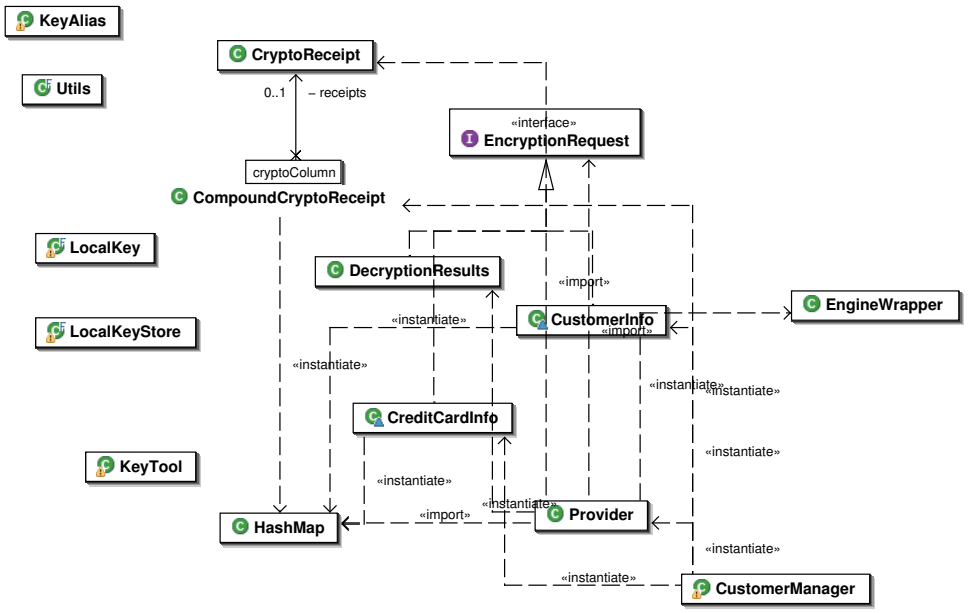


Figure 7.25: CryptoDB: class diagram, extracted using Eclipse UML (Omondo 2006).

Figure 7.26: CryptoDB: flat object graph extracted using PANGAEA (Spiegel 2002).

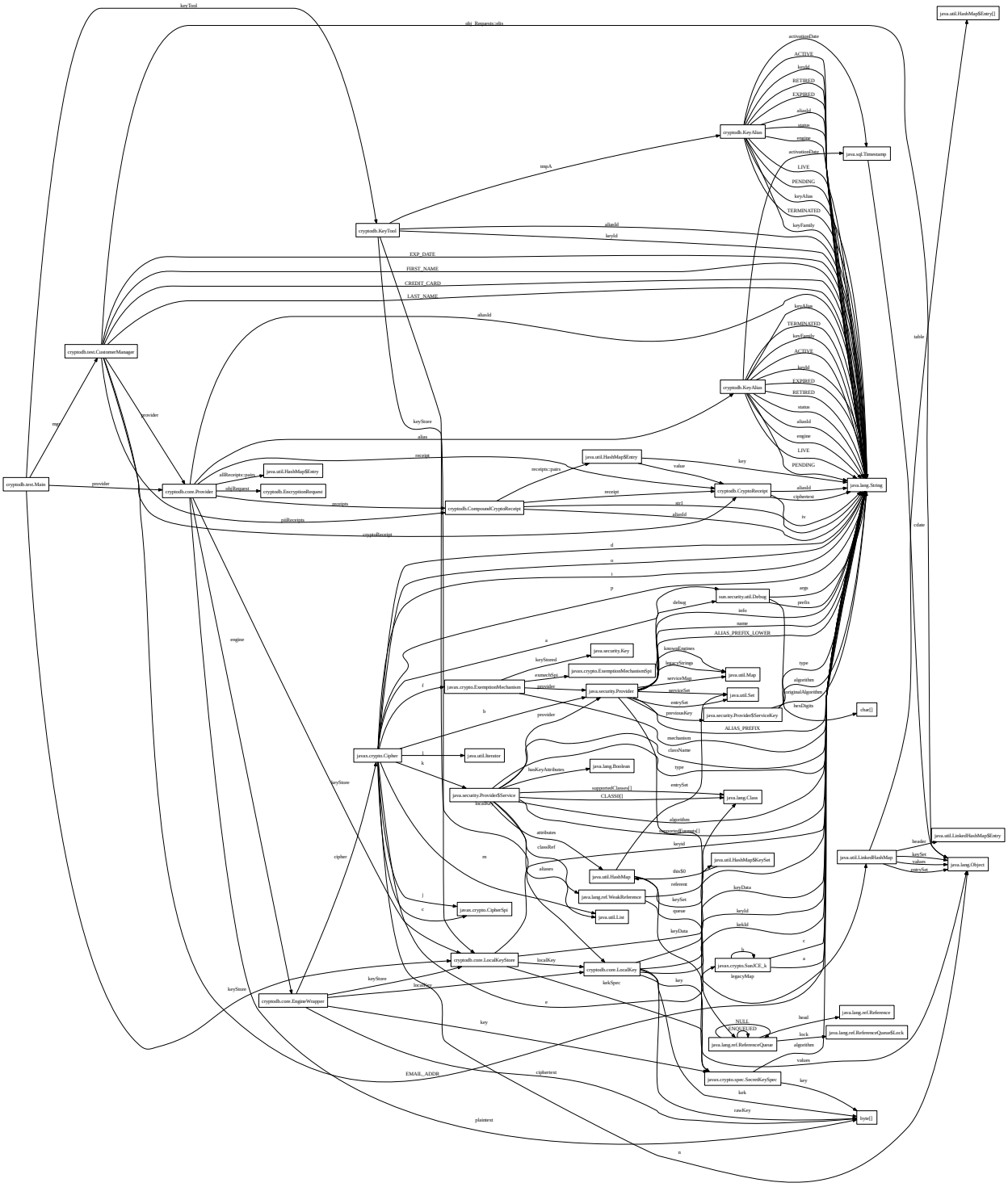


Figure 7.27: CryptoDB: flat object graph extracted using WOMBLE (Jackson and Waingold 2001).

```
1  class LocalKeyStore<KEYID> {
2    private domain OWNED, KEYDATA;
3    public domain KEYS;
4    link KEYS -> KEYID, KEYS -> KEYDATA, OWNED -> KEYS;
5    assume OWNER -> KEYID;
6    private OWNED List<KEYS LocalKey<KEYID,KEYDATA>> keys;
7
8    public unique List<KEYS LocalKey<...>> getKeys() {
9      unique List<KEYS LocalKey<...>> copy = copy(keys);
10     return copy;
11   }
12 }
13 class LocalKey<KEYID,KEYDATA> {
14   assume OWNER -> KEYID, OWNER -> KEYDATA;
15   private KEYDATA String keyData; // encrypted key
16   private KEYID String keyId; // encrypted key id
17   ...
18   private OWNER SecretKeySpec key; // Make peer to self
19 }
```

Figure 7.28: CryptoDB: `LocalKeyStore` and `LocalKey` annotations.

7.8.3 Adding Annotations

I added ownership domain annotations to CryptoDB to specify, within the code, object encapsulation, logical containment and architectural tiers, as discussed earlier. The annotations define two kinds of object hierarchy, logical containment and strict encapsulation.

Logical containment. As an example of logical containment in CryptoDB, `LocalKeyStore` declares a public domain, `KEYS`, to hold `LocalKey` objects (line 3 in Fig. 7.28).

Strict encapsulation. As an example of *strict encapsulation* in CryptoDB, `LocalKeyStore` stores the list of `LocalKey` objects, keys, in a private domain, `OWNED` (line 6). As a result, the accessor `getKeys` must return a shallow copy of the list, and cannot return an alias (line 8 in Fig. 7.28).

Domain parameters. I defined on the class `LocalKey` the `KEYID` and `KEYDATA` domain parameters (line 13). In turn, `LocalKeyStore` takes a `KEYID` domain parameter (line 1). For example, `LocalKeyStore` binds its local domain `KEYDATA` to `LocalKey`'s `KEYDATA` parameter (line 6).

Top-level domains. I organized instances of the core CryptoDB types into four top-level domains, as follows (Fig. 7.31, 7.32):

- `CONSUMERS`: has `CustomerManager`, and `EncryptionRequests`, such as `CustomerInfo` and `CreditCardInfo`;

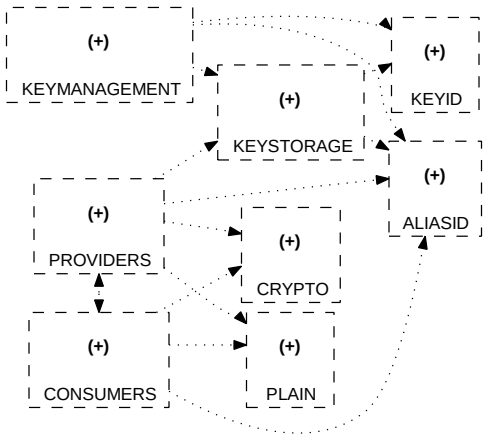


Figure 7.29: CryptoDB: Level-0 OOG with `String` objects.

- `PROVIDERS`: has `Provider`,`EngineWrapper`;
- `KEYSTORAGE`: has `KeyAliases` and `LocalKeyStore`;
- `KEYMANAGEMENT`: has a `KeyTool` object.

Nested domains. For several classes C_i, I also defined one or more nested domains D_i, which I refer to using the C_i::D_i notation:

- `CustomerManager`::`RECEIPTS` has `CryptoReceipts`;
- `LocalKeyStore`::`KEYS` has instances of `LocalKey`, `SecretKeySpec`, etc. (Fig. 7.33). In contrast, the private `OWNED` domain contains a `List` of `LocalKeys` .
- `Provider`::`RCPTMGR` has `CompoundCryptoReceipt` objects;

Refining the annotations. I iterated the process of adding the annotations a few times. In one such refinement, I wanted to reason about `String` objects. In the previous case studies, `String` objects were uninteresting, and annotated with `shared`. However, when reasoning about security, `String` objects become interesting. Indeed, in CryptoDB, much communication takes place through `Strings`. To better understand this communication, we declared different domains for plain-text (`PLAIN`), encrypted (`CRYPTO`), alias identifier (`ALIASID`), and key identifier (`KEYID`) `Strings`. In particular, the annotation typechecker checks that these `Strings` are not assigned to each other, a perfectly valid operation in Java.

For example, Fig. 7.29 shows only the top-level domains and summarizes the field references between objects in those domains using dotted edges. However, when analyzing conformance later, we simplified the OOG by binding all the additional parameters for `PLAIN`, `CRYPTO`, etc., to the `shared` domain. This required changing only the binding of these domain parameters in the top-level class, and changing a few lines of annotations in the top-level class.

An object graph showing explicit top-level domains for the different kinds of `Strings` is in Fig. 7.30.

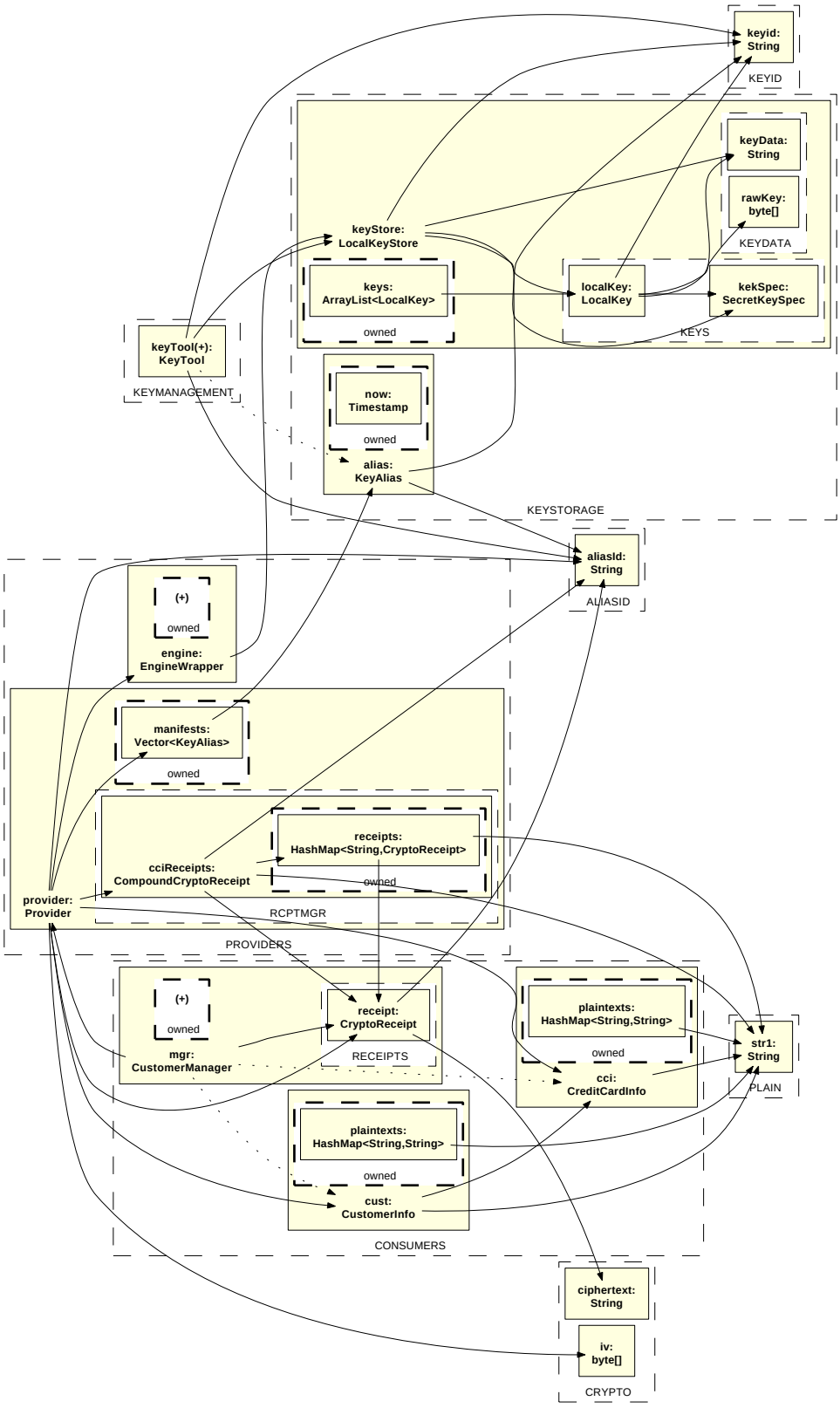


Figure 7.30: CryptoDB: OOG with `String` objects.

```
interface EncryptionRequest<PLAIN> {
  unique Map<PLAIN String, PLAIN String> getPlaintexts();
}
class DecryptionResults<PLAIN> implements EncryptionRequest<PLAIN> {
  private domain OWNED;
  OWNED Map<PLAIN String, PLAIN String> plaintexts = new ...;
  unique Map<...> getPlaintexts() {
    return copy(plaintexts); // Return copy of field
  }
}
class CompoundCryptoReceipt<RECEIPTS,PLAIN,CRYPTO,ALIASID> {
  private domain OWNED;
  OWNED Map<PLAIN String,RECEIPTS CryptoReceipt> receipts = new ...;
}
class CryptoReceipt<CRYPTO,ALIASID> {
  CRYPTO String ciphertext;
  CRYPTO String iv;
  ALIASID String aliasId;
}
```

Figure 7.31: CryptoDB: annotation excerpts.

7.8.4 Extracting Object Graphs

I then used ArchRecJ to extract an OOG from the annotated CryptoDB code (Fig. 7.35). For example, inside the `Provider`'s `RCPTMGR` domain, a `CompoundCryptoReceipt` encapsulates a `HashMap` that maps `String` to `CryptoReceipt` objects. Separately, each `EncryptionRequest` inside the `CONSUMERS` domain has a `HashMap` that maps `Strings` to `Strings`.

Abstraction by types. An object graph without abstraction by types shows separate `CustomerInfo` and `CreditCardInfo` objects (Fig. 7.35). Because the target architecture has one such component, I used abstraction by types to make the CryptoDB OOG merge objects of type `CustomerInfo`, and `CreditCardInfo` in the `CONSUMERS` domain, because their classes implement the `EncryptionRequest` interface (Fig. 7.31). To do so, I added the interface `EncryptionRequest` to the list of design intent types.

Hierarchy. Fig. 7.34 shows the top-level domains and the objects directly inside them, with their substructure collapsed, after binding all the domain parameters containing `Strings` to `shared`. In Fig. 7.35, we manually expanded the substructures of `mgr`, `provider`, `engine`, etc. Here, we collapsed the substructure of `keyStore` (which appears in Fig. 7.33). We also manually collapsed the private domain `OWNED` inside `keyStore`, which now appears as `OWNED(+)`.

```
class CreditCardInfo<PLAIN> implements EncryptionRequest<PLAIN> {
  public unique Map<...> getPlaintexts() {
    unique Map<PLAIN String, PLAIN String> map = new ...;
    map.put(CustomerManager.CREDIT_CARD, creditCard);
    ...
    return map;
  }
}
class CustomerManager<CONSUMERS,PROVIDERS,PLAIN,CRYPTO,ALIASID...> {
  public domain RECEIPTS;
  PROVIDERS Provider<CONSUMERS,PLAIN,CRYPTO,ALIASID,RECEIPTS...> prov;
  void testEncrypt() {
    CONSUMERS CreditCardInfo<PLAIN> cci = new CreditCardInfo();
    prov.RCPTMGR CompoundCryptoReceipt<...> cciRcpts = prov.encrypt(cci, "cci");
  }
  void testDecrypt() {
    prov.RCPTMGR CompoundCryptoReceipt<...> pii = new ...;
    RECEIPTS CryptoReceipt<CRYPTO,ALIASID> r1 = new ...;
    pii.addReceipt(FIRST_NAME, r1);
    CONSUMERS DecryptionResults<PLAIN> piiPlaintexts = prov.decrypt(pii);
  }
}
class System {
  domain CONSUMERS,PROVIDERS,KEYMANAGEMENT,KEYSTORAGE...;
  KEYSTORAGE LocalKeyStore<...> store = new LocalKeyStore();
  KEYMANAGEMENT KeyTool<KEYSTORAGE...> tool = new KeyTool(store);
  CONSUMERS CustomerManager<...> mgr = new CustomerManager(store);
}
```

Figure 7.32: CryptoDB: annotation excerpts (continued).

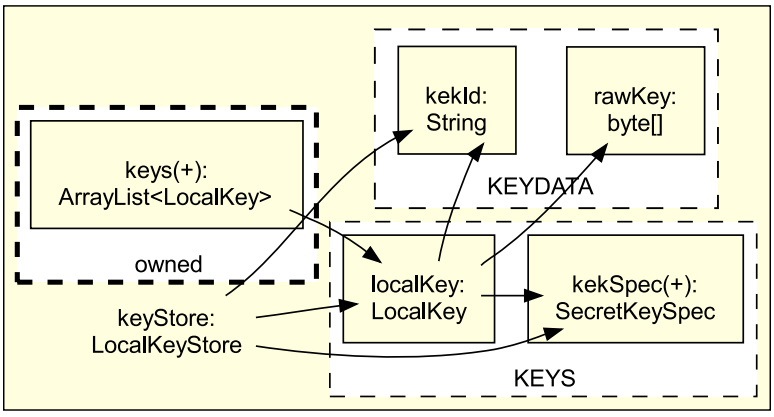


Figure 7.33: CryptoDB: `LocalKeyStore` OOG.

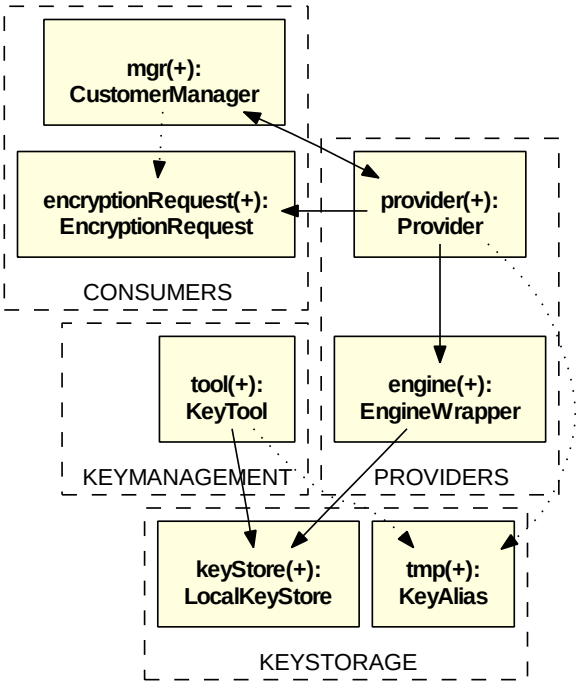


Figure 7.34: CryptoDB: Level-1 OOG without `String` objects.

7.8.5 Abstracting into Built Architecture

I iterated the process of adding the annotations and extracting OOGs until the OOG had roughly similar tiers, a similar hierarchical decomposition, and a similar number of components in each tier, when visually compared to the target architecture. I then used ArchCog to abstract an extracted object graph into a C&C architecture represented in Acme (Fig. 7.36).

7.8.6 Modeling the Target Architecture

We designed a target architecture using Acme, basing it largely on the available DFDs (Section 7.8.2.1). We represented the DFD processes and data stores using components. We used the Acme representation feature to include subarchitectures corresponding to second-level DFDs. We used Acme groups, depicted with dashed lines, to partition the architecture into broad areas of responsibility.

We added directional connectors based on the information in the book by (Kenan 2006). In many cases, the points-to connectors were the reverse of the data flow connectors in the DFDs.

We went through a process of iteration to get the architecture right. This was due in large measure to the ways in which the implementation departed from the architecture. The implementation, in our case, was a demonstrative implementation found in a security book, not a fully faithful implementation of the design. In particular, the implementation was simplified in many respects. For instance, Kenan identifies in principle a number of subcomponents of the cryptographic provider: an initializer, an encoder, a receipt manager, an engine interface, and others (Kenan 2006, §6.1). In the implementation, the provider was nearly monolithic; few of these

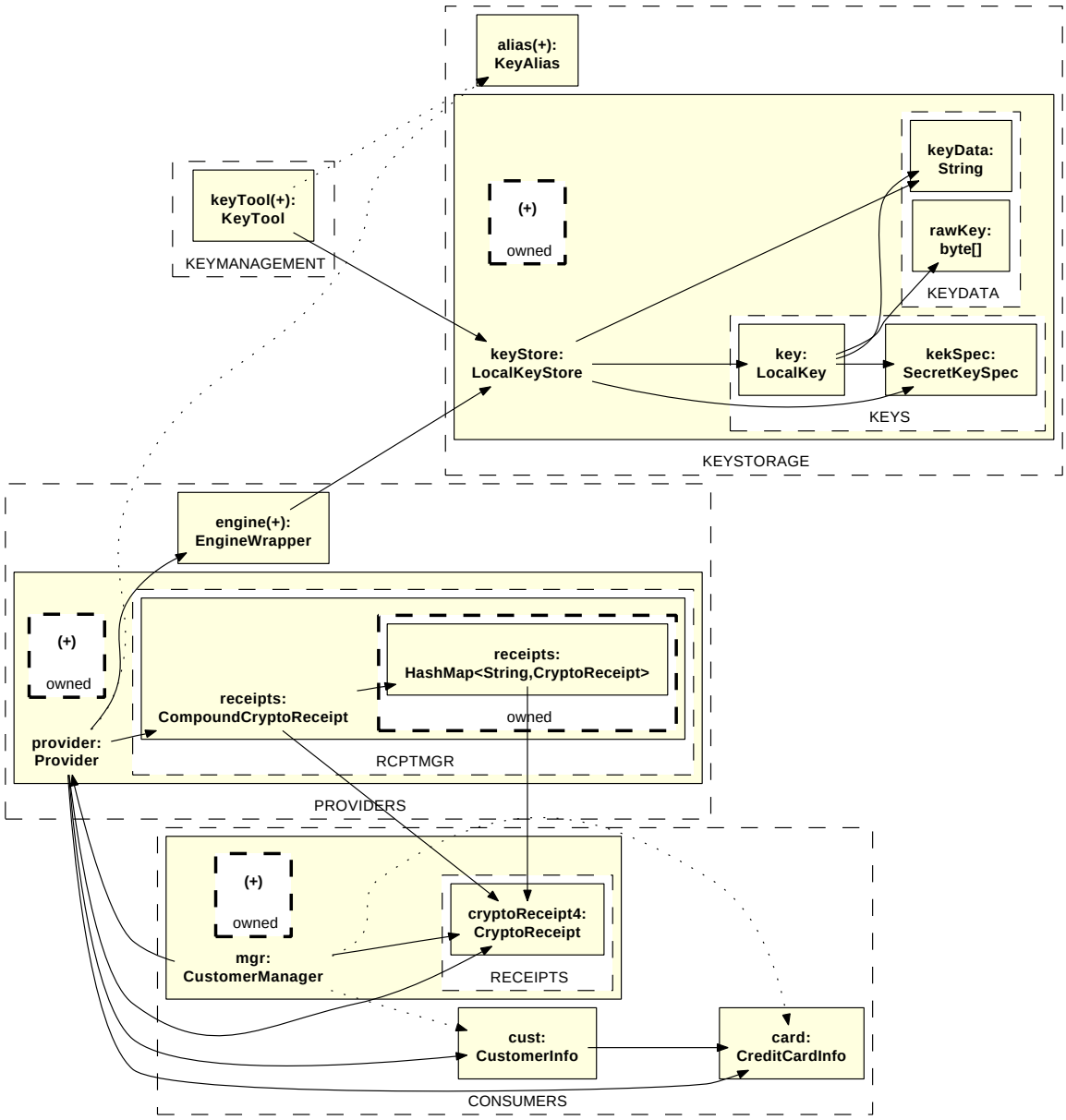


Figure 7.35: CryptoDB: Level-2 OOG, after binding top-level domains for `String` to `shared`.

distinct responsibilities were actually allocated to separate objects. We had to modify our target architecture to accommodate the casual way in which the implementation realized the described architecture. Had we not done so, we would have had to deal with these discrepancies later while analyzing conformance. In a system in which the implementation more faithfully realized the design, less iteration would be necessary.

This iteration was partly due to the mismatch between conceptual and implementation-level architectures. In Acme, a component is just a transparent view of a more detailed decomposition specified by the representation of that component (Section 5.4.2, Page 189). In both the OOG and the abstracted built architecture, a component collapses one or more objects that constitute

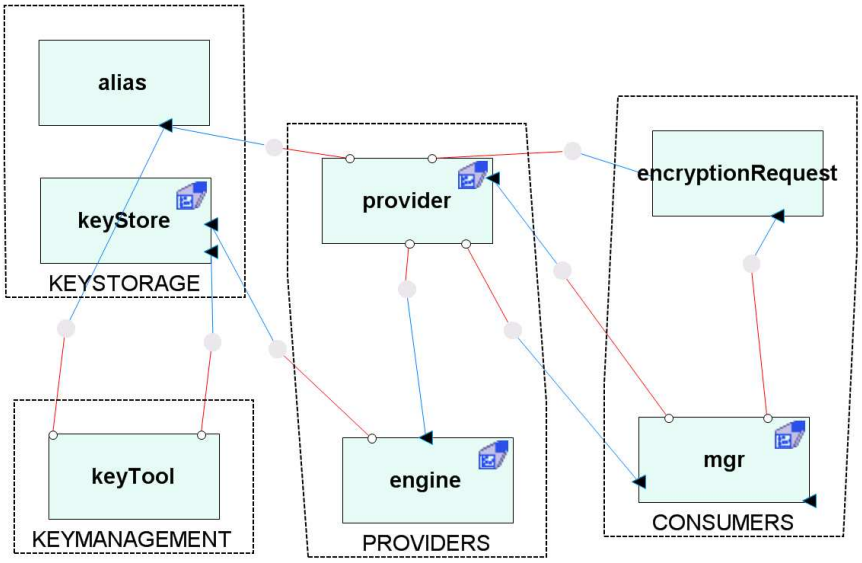


Figure 7.36: CryptoDB: built architecture in Acme.

its parts, according to their ownership and type structures.

In general, developers do not use hierarchical decomposition rigorously in DFDs. But in SCHOLIA, logical containment can push almost any object underneath any other object in the ownership hierarchy. This allows a developer to use annotations to control the system decomposition in the OOG.

Another change we made in the process of iteration was to exclude the external interactors from the target architecture. Although useful for showing the endpoints of a system during threat modeling, they did not correspond to any code elements, since they were external to the system. We could leave the external interactors in the target architecture, but they would always show up as absences in the conformance view, thus increasing the noise level.

While iterating the process of adding the annotations and extracting OOGs, we determined the similarity between the OOG and the target architecture by visual inspection. The CryptoDB target architecture we converged on is in Fig. 7.37.

7.8.7 Analyzing Conformance

I then analyzed the communication integrity of the CryptoDB target architecture, and established the traceability between the target architecture and the code. I used ArchConf to create a *conformance view* of the target architecture (Fig. 7.38), which shows convergences, divergences, and absences, and has traceability to the code.

Renames. Because SCHOLIA uses a structural comparison algorithm to compare the built and designed architectures, it was able to handle the naming discrepancies between the target architecture and the implementation, e.g., `KeyManager` versus `KeyTool`.

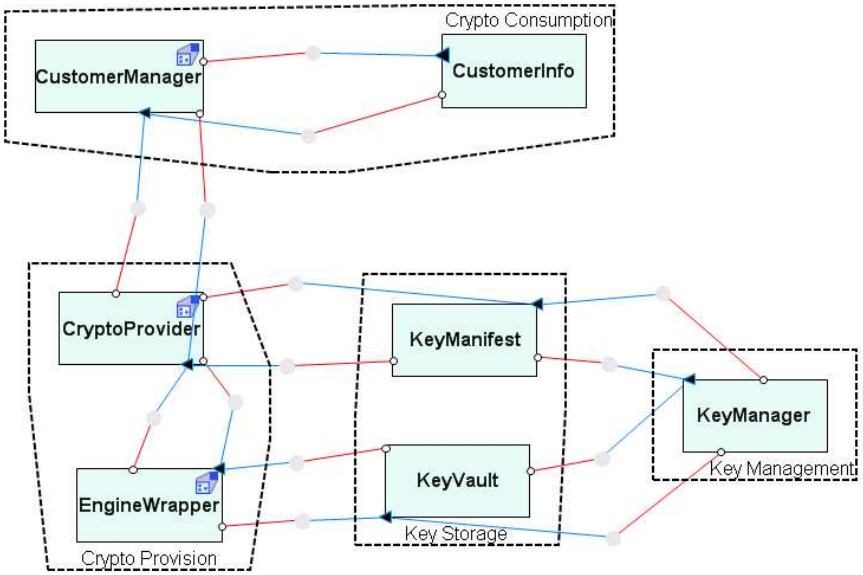


Figure 7.37: CryptoDB: target architecture in Acme.

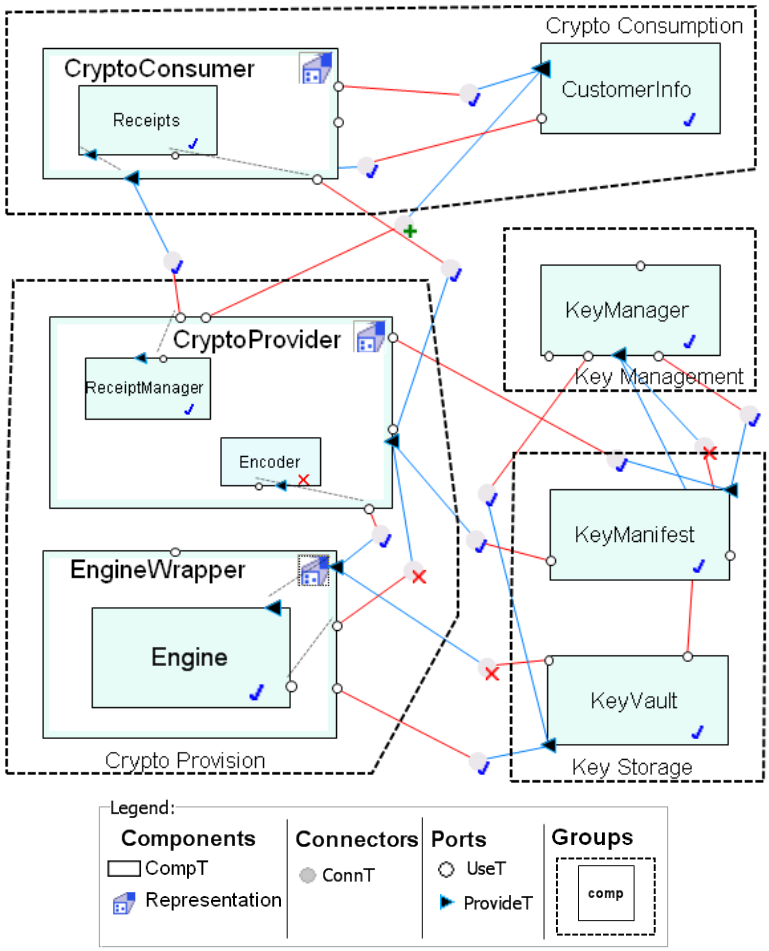


Figure 7.38: CryptoDB: conformance view in Acme. The representation of some components is inlined.

Conformance findings. Overall, as the large number of convergences indicates, the top-level components in the target architecture (based on a Level-1 DFD) and the implementation were mostly consistent (Fig. 7.38).

Drilling down into the representations of the some of the top-level components revealed more interesting differences. For example, the Level-2 DFD (Fig. 7.23) shows an Encoder component inside the Provider component. However, the implementation represents the Encoder's functionality using a helper class `Utils`, which is never instantiated. Hence, the conformance view shows an absence. One way to resolve this absence is to modify the code to instantiate a singleton `Utils` object, which would not affect the system's behavior. Alternatively, we could use a "virtual field" annotation that indicates an object allocation, to force the OOG to show an instance of the `Encoder` class.

In the process of modeling the target architecture, we confronted a number of architecture–implementation discrepancies of this nature. We ultimately dealt with them, in most cases, by modifying the target architecture to match the implementation. This was necessary because of the departures that the CryptoDB implementation made from the architecture. Had we not reconciled the differences in the target architecture, we would have had more noise to sort through while analyzing conformance. Naturally, distinguishing between deliberate departures from the architecture and genuine architecture violations requires careful judgment. However, we view it as a strength of SCHOLIA that architects have the opportunity to exercise their judgment in this way to forestall uninteresting violation reports from the tool.

In other cases, we refined the annotations. For instance, we had initially modeled all instances of `CryptoReceipt` and `CompoundCryptoReceipt` in a `RECEIPTS` domain inside the `CustomerManager`. As a result, the analysis flagged the `ReceiptManager` inside the `CryptoProvider` as an absence. Then we looked more carefully at how the `Provider` and the `CustomerManager` exchanged these objects. This led us to define a `RCPTMGR` domain inside `provider` for `CompoundCryptoReceipts`, and left the `CryptoReceipts` in the `RECEIPTS` domain inside `mgr` (Fig. 7.31).

7.8.8 Enforcing Code-Level Constraints

We then added to CryptoDB domain links to specify explicit policies that govern how a domain can reference objects in other domains (Section 2.3.2, Page 40).

For example, in CryptoDB, a `LocalKey` *assumes* that its owning domain can access the `KEYID` and `KEYDATA` domain parameters. In turn, when a `LocalKeyStore` instantiates a `LocalKey`, and binds `KEYID` and `KEYDATA` to `KEYID` and `KEYS`, respectively, `LocalKeyStore` must satisfy those permissions. For the first one, it declares a *domain link* from `KEYS` to `KEYID` (line 4). For the second one, it links `KEYS` to `KEYDATA`.

We defined domain links and assumptions and typechecked them. The resulting domain link declarations in the top-level class were largely expected. As can be seen in Fig. 7.29, there are bidirectional links between `PROVIDERS` and `CONSUMERS`. But the links are unidirectional from `PROVIDERS` and `KEYMANAGEMENT` to `KEYSTORAGE`. Of course, there are no links from `CONSUMERS` to `KEYSTORAGE`. Note that domain link permissions are not transitive.

```
class Provider<REQUESTS,KEYSTORAGE...> {
  assume OWNER->KEYSTORAGE;
  KEYSTORAGE LocalKeyStore<KEYID> keyStore; // (1)
  OWNER EngineWrapper<KEYSTORAGE...> engine;

 Provider(KEYSTORAGE LocalKeyStore<KEYID> store) {
   // Inject architectural violation
   this.keyStore = store; // (2)
   this.engine = new EngineWrapper(store);
 }
}
```

Figure 7.39: CryptoDB: injected architectural violation.

7.8.9 Enforcing Architectural Constraints

We also wrote architectural constraints to express restrictions on the communication allowed in the CryptoDB architecture. Then, we formalized these constraints and added them to the CryptoDB target architecture. Some of the constraints include:

1. `KeyManager` should not connect to `EngineWrapper`;
2. `KeyVault` should not point to `KeyManifest`;
3. Only `KeyManager` and `EngineWrapper` should have access to `KeyVault`.

All these constraints reflect our understanding of the security requirements of the target architecture, and indeed they are all roughly derived from commentary in Kenan's book (Kenan 2006). For example, constraint 3 is an adaptation of the following remark: "Access to the key vault [...] should be granted to only security officers and the cryptographic engine" (p. 71). The key manager is the architectural agent that security officers use, hence we arrive at constraint 3.

We formalized the above constraints using the Acme predicate language (Monroe 2001), as follows:

1. `forall c : Component in KeyManagement.MEMBERS |`
   `!connected(c, EngineWrapper)`
2. `!pointsTo(KeyVault, KeyManifest)`
3. `forall c : Component in self.COMPONENTS |`
   `pointsTo(c, KeyVault) -> c.label=="KeyManager"`
   `or c.label=="EngineWrapper"`

The full Acme specification of the CryptoDB target architecture, including the architectural style and the definition of the `pointsTo` predicate above, is in Appendix B.

Constraint violations. Once we added the constraints to the target architecture, we used the AcmeStudio tool to verify them. Due to the traceability SCHOLIA established between the architecture and the code, we can have confidence that the implementation meets these constraints.

To further validate our approach, we modified the CryptoDB code, injecting a manufactured architecture violation to confirm that our constraints would catch it. Specifically, we coupled the `Provider` and the `LocalKeyStore` (Fig. 7.39). According to constraint 3 above, the `Provider` is not allowed to point to the `LocalKeyStore` in this way. In the architecture, access to the

KeyVault is highly restricted due to the sensitivity of the contents.

When we modified the code in this way and ran our analysis, the conformance view showed an additional divergence between provider and keyVault, and the predicate raised a warning about the architectural violation in the conformance view. In addition, the domain link checks alone would not have caught this violation. Both engine and provider are peers in the same PROVIDERS domain (Fig. 7.34). So, there must already be a domain link from PROVIDERS to KEYSTORAGE for engine to access the key vault. But we still do not want provider to access the key vault.

7.8.10 CryptoDB Discussion

The CryptoDB case study demonstrates that SCHOLIA can relate, entirely statically, a security runtime architecture to a program written in a widely used object-oriented language, using annotations. Such an approach can increase the effectiveness of reasoning architecturally about the security of existing systems, because it ensures that the architecture is a faithful representation of the code, which is ultimately the most reliable and accurate description of the built system. Of course, many approaches identify security vulnerabilities directly at the code level, without requiring ownership annotations, or following the SCHOLIA approach. However, architectural analysis matches the way experts reason about security or privacy better than a purely code-based strategy, as indicated by the well-established threat modeling process.

Architectural security analysis. Various architectural-level security analyses have been proposed (Moriconi et al. 1997; Deng et al. 2003). For example, UMLsec (Jürjens 2004) extends UML with secrecy, integrity and authenticity, to allow analyzing security weaknesses at the design level. However, UMLsec achieves conformance between the architecture and the implementation using code generation, code analysis, and test-sequence generation. Code generation, while potentially guaranteeing the correct refinement of an architecture into an implementation, is often too restrictive to be fully adopted on a large scale and cannot account for legacy code. One could use SCHOLIA to analyze an existing system, after the fact, by adding annotations to the code.

Similarly, SecureUML (Lodderstedt et al. 2002) recommends a model-driven approach in which security constraints are imposed on a model that is later elaborated into code. Of course, like all model-driven approaches, SecureUML is useful only for construction of new systems, not for analysis of existing implementations. SCHOLIA is appropriate for use on existing code, requiring only annotations. Another difference is that SecureUML is based on a code architecture, leaving other views for future work.

Code-level analyses. Many code-level analyses can identify security vulnerabilities by static analysis directly over the code. SCHOLIA complements, and does not supplant these code-level analyses. Moreover, the traceability between a security architecture and the code that SCHOLIA derives can benefit other static analyses. Until now, due to the lack of traceability, much of the security design intent generated during threat modeling has not been accessible to other code quality tools. For instance, a static analysis checking for buffer overruns, e.g., (Hackett et al.

2006), can use this traceability to assign to its warnings more appropriate priorities based on a more holistic view of the system.

Security testing. Analysis offers substantial benefits beyond those of testing alone. Perhaps most significantly, since SCHOLIA is based on static analysis, it can reveal information about all possible runs of a program, while testing is limited to a small number of runs. This difference is particularly important in the security domain. Similar to testing is dynamic conformance analysis, which instruments and monitors a system (Sefika et al. 1996b; Schmerl et al. 2006). We discuss checking conformance using dynamic analysis further in Section 8.9.2.1 (Page 301)

Design enforcement. Many approaches can enforce local, modular, code-level constraints, e.g., JavaCop (Andreae et al. 2006), SCL (Hoover and Hou 2006). SCHOLIA is complementary and can enforce structural constraints on the global runtime architectural structure. As we discussed in Sections 6.5, 7.8.9, documenting an extracted architecture in an ADL enables first-order logic predicates to enforce global constraints on the architecture (Monroe 2001).

Conformance to a style. Many approaches can analyze the conformance of an architecture to an architectural style, but assume that the architecture is extracted somehow. Thus, such approaches can be seen as addressing the problem of *horizontal conformance* (Ducasse and Pollet 2009), rather than *vertical conformance*. For instance, (Medvidovic and Jakobac 2006) check the conformance of an implementation with respect to an architectural style, but manually relate the designed and the built architectures. SCHOLIA is an integrated approach to analyze both vertical and horizontal conformance.

7.9 Discussion

In this section, I discuss the SCHOLIA evaluation. I first discuss the *external validity*, to what extent the results can be generalized, then revisit the research questions.

7.9.1 External Validity

Can SCHOLIA find architectural violations in other systems? To date, I have evaluated the end-to-end SCHOLIA approach on four systems, Aphyds, JHotDraw, HillClimber and CryptoDB. I did not obtain a designed runtime architecture for LbGrid, so I could not analyze its conformance. For most systems, the challenge is to find a designed runtime architecture that is documented, or to have access to a developer's architectural intent.

In all the architectures we analyzed, SCHOLIA found omitted components, connectors or entire sub-architectures. For example, the JHotDraw designed architecture omitted several components that were a later addition to support undoing commands (Section 7.6).

Can SCHOLIA analyze architectures that specify fine-grained object structures or multiplicities? An OOG and its abstracted C&C view provide architectural abstraction by merging

equivalent instances in a domain or tier. So, there will be diagrams that show very fine-grained object structures, for which SCHOLIA's abstraction would be too coarse. Similarly to most static object diagrams, SCHOLIA does not provide any precision regarding multiplicities.

Would an outside developer understand the SCHOLIA technique? Until there are better tools for adding annotations, SCHOLIA does not have the characteristic of Reflexion Models that third-party users can run on large bodies of code (Murphy et al. 2001). As a result, a study with an outside developer would be difficult given the nature of the approach.

We did, however, conduct a field study, and confirmed that an outside professional programmer understood abstraction by ownership hierarchy and by types (Section 4.8).

It is true that iteratively improving the annotations and fine-tuning the abstraction and following analysis steps in the tool chain may be a challenge. However, this situation is not unique to SCHOLIA. For example, previous work on code architectures using semi-automated clustering algorithms, required engineers to spend significant effort fine-tuning the clustering parameters to derive a good match (Christl et al. 2005). In SCHOLIA, a developer does not rely on a tool's hard-coded heuristics but controls the architectural abstraction using annotations. As the evaluation showed, I was able to refine the annotations to get a better match, without changing the code.

Is SCHOLIA more *lightweight* than other static conformance approaches? For example, *is adding ownership annotations to an existing system less invasive than re-engineering it to ArchJava to expose its architecture?* Our preliminary evidence showed that to be the case (Abi-Antoun et al. 2007a). The annotations, unlike ArchJava, do not change the system's runtime semantics, and support common object-oriented idioms, such as passing references to objects. For example, an ArchJava `component class` cannot have `public` fields. When using ownership annotations, such legal Java fields can be placed in public domains. (Aldrich et al. 2002c) added ownership types to the model part of Aphyds (3.5 KLOC) in 4 hours, a quarter of the time they spent re-engineering that same part to ArchJava.

To more reliably estimate the annotation effort, I conducted a week-long on-site field study. I spent 35 hours adding annotations and extracting OOGs from the 30-KLOC LbGrid module (WARN is still high). Based on our previous experience with ArchJava (Abi-Antoun and Coelho 2005; Abi-Antoun et al. 2007a), I could not have re-engineered LbGrid to ArchJava in the same few days that it took me to add the annotations, even after accounting for possible tool and language familiarity. Thus, adding annotations to an existing system seems more lightweight than re-engineering the system to use an extended language like ArchJava.

Would SCHOLIA work with an ownership type system other than ownership domains? In principle, SCHOLIA could use a type system that assumes a single *context* per object (Clarke et al. 1998). There is, however, a crucial expressiveness advantage in ownership domains that can reduce the number of objects in the top-level domains. In an *owner-as-dominator* type system, any access to a child object must go through its owning object (Clarke et al. 1998). As a result, this forces more objects to be peers. When annotating arbitrary object-oriented code after the fact, it

is easier to use *logical containment* with public domains, rather than the strict *encapsulation* of private domains, and both can reduce the number of objects in the top-level domains.

Can SCHOLIA scale to big systems? Architectural extraction is most useful for large systems. In general, the tools for analyzing the runtime architecture are not as mature as the tools for the code architecture. For comparison, the closest prior work that used annotations to extract object models that provide architectural intent was evaluated on one 1,700-line system (Lam and Rinard 2003). In contrast, (Murphy et al. 2001) evaluated Reflexion Models on million-line systems.

As a type-based technique that requires developers to specify architectural intent using annotations, SCHOLIA is currently prohibitively costly for systems with millions of lines of code. Scaling SCHOLIA to large systems requires better tools for inferring the annotations. Alternatively, developers can be required to add and update the annotations during development.

7.9.2 Research Questions (Revisited)

In this section, I discuss how well the evaluation answered the research questions (Section 7.2).

RQ1 – Extraction: In practice, I was able to extract an object graph that expresses architectural intent and conveys architectural abstraction by ownership hierarchy and by types. Indeed, I was able to reduce the number of top-level objects compared to a flat object graph, and not display low-level objects. In addition, I was able to achieve a similar hierarchical decomposition and a similar number of objects and domains at each hierarchy level, when visually compared to a target architecture. However, there still a few annotation warnings, so the extracted object graph is not guaranteed to be sound.

RQ2 – Abstraction: In practice, I was able to use ArchCog to abstract a hierarchical object graph into a sensible runtime architecture represented as a C&C view in AcmeStudio. In most cases, I used the default options for abstracting an object graph.

RQ3 – Comparison: In practice, the structural comparison was able to meaningfully compare the built architecture extracted from the implementation to a designed architecture. In only a few cases, I had to manually force or prevent matches between the view elements.

RQ4 – Checking: In practice, the conformance analysis was able to match the built and the designed architectures, display a readable conformance view, enable tracing a finding to the code, and compute sensible conformance metrics. The conformance view highlighted communication that is present in the implementation but not in the designed architecture, and vice versa. In practice, the conformance analysis did not generate too many false positives. And I was able to trace from the conformance view to the right code locations. In particular, with good annotations, the conformance analysis did not generate a conformance view which consisted of an unreadable fully connected graph, which had much noise that I had to wade through.

7.9.3 Performance

Table 7.5 shows a performance summary.

Table 7.5: Performance measurements of the conformance analysis. CCM is the core conformance metrics. LOC measures the lines of code. OOG and SYNC are the OOG extraction and structural comparison times, respectively, measured in minutes and seconds on an Intel Pentium 4 (3 GHz) with 2 GB of memory. WARN measures the remaining warnings.

System	CCM	LOC	OOG	SYNC	WARN
JHotDraw	54 %	15,000	1:22	1:44	60
HillClimber	83 %	15,000	1:08	0:54	42
Aphyds	29 %	8,000	0:37	2:05	72

7.9.4 Evaluation Critique

Our evaluation of the conformance analysis shares several limitations with our evaluation of the object graph extraction, discussed in Section 4.10.2 (Page 175), and suffers from the following additional limitations.

Target architecture. The process of deriving a reference or target architecture is a research topic in its own right. There are potentially several issues with the target architectures we used in the evaluation of the conformance analysis.

- **Aphyds** (Section 7.5): the original Java developer designed the target architecture. The edges mixed control flow and data flow information. The diagram had some system decomposition information;
- **JHotDraw** (Section 7.6): we had access to an abstracted class diagram, but did not have a target runtime architecture designed by one of the original developers.
- **HillClimber** (Section 7.7): one of the original developers designed a target runtime architecture that showed only the top-level components and lacked hierarchical decomposition;
- **CryptoDB** (Section 7.8): the original Java developer documented various Data Flow Diagrams that showed data flow edges, and used system decomposition informally.

Communication integrity. The remaining annotation warnings in the subject systems weaken the claims that the extracted object graphs are sound. As a result, the analyzed target architectures of those systems may still not satisfy the communication integrity principle.

7.10 Summary

In this chapter, I evaluated SCHOLIA on several real object-oriented systems. The evaluation showed that SCHOLIA can be applied to an existing system while changing only annotations in the code. In all the architectures we analyzed, SCHOLIA found interesting architectural structural differences between the implementation and the target architecture. In addition, SCHOLIA established traceability, after the fact, between the target architecture and the code.

Credits

Jeffrey Barnes read carefully Kenan's book (Kenan 2006), iterated the process of designing the CryptoDB target architecture, defined the structural constraints, and tracked down several AcmeStudio bugs. He also contributed to the writing of the paper (Abi-Antoun and Barnes 2009a) and online appendix (Abi-Antoun and Barnes 2009b).

Acknowledgements

The author would like to thank Bradley Schmerl for his help with Acme and AcmeStudio. In addition to the thesis committee, David Garlan and Mary Shaw gave us very useful feedback.

Chapter 8

Related Work

SCHOLIA builds on a rich body of research in the area of object-oriented design diagrams (Section 8.1), software architecture (Section 8.2), ownership type systems (Section 8.3), static analysis of the runtime structure (Section 8.4), dynamic analysis of the runtime structure (Section 8.5), architectural extraction (Section 8.6), architectural comparison (Section 8.7), built-in conformance (Section 8.8), after-the-fact conformance analysis of architectures (Section 8.9) and traceability analysis (Section 8.10).

8.1 Object-Oriented Design Diagrams

The structure of an object-oriented system is commonly described using an object-oriented modeling notation, such as the standard Unified Modeling Language (UML) (Rumbaugh et al. 1998).

Class diagrams vs. object diagrams. Most object modeling notations support both class diagrams which show the type structure of the system, and object diagrams which represent its runtime structure.

Static object diagrams vs. dynamic object diagrams. In Chapter 1, I adopted the terminology of (Tonella and Potrich 2004) and distinguished between *static object diagrams* and *dynamic object diagrams*. This distinction is also helpful to organize previous work, and relate SCHOLIA to that work.

A *static object diagram* shows all possible objects and relations between those objects, across all program runs, and is extracted by static analysis over the code. A *dynamic object diagram*, which is recovered using a dynamic analysis, shows the objects and the relations that are created during a specific system execution (Tonella and Potrich 2004).

UML. Paradoxically, the UML specification (version 1.3) seems partly to blame for the lack of attention paid to object diagrams, and relegating them to play a smaller role in UML (the emphasis is mine): "An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. *The use of object diagrams is fairly limited, mainly to*

show examples of data structures. Tools need not support a separate format for object diagrams. Class diagrams can contain objects, so a class diagram with objects and no classes is an 'object diagram' " (OMG 2008).

UML and ownership. SCHOLIA is not the first approach to represent ownership information in an object-oriented design diagram. For instance, (Liu and Milanova 2007) augment a UML class diagram with ownership information. However, they assume an ownership model that does not have ownership parameters, which is less flexible than the type system SCHOLIA uses. In addition, displaying object-level ownership on a class diagram is problematic. Typically, a class diagram shows only one box for a class `List`. It is unclear how such an approach can display different instances of a `List` object that are owned or strictly encapsulated by different instances of some other class.

Program understanding. Many researchers have long recognized the importance of understanding the runtime structure of a system. For example, (Kirk et al. 2006) state that object-oriented frameworks pose particular program understanding challenges, and emphasize that "understanding the dynamic behavior of a framework is more challenging, particularly given the separation of the static and dynamic perspectives in the object-oriented paradigm". (Shull et al. 2000) concur that both "the static and dynamic structures must be understood and then adapted to the specific requirements of the application [...] For a developer unfamiliar with the system to obtain this understanding is a non-trivial task. Little work has been done on minimizing this learning curve".

We believe that SCHOLIA, which can help a developer extract from an implementation a runtime view for system understanding purposes, is a step in the right direction.

Empirical evaluation of design diagrams. Several researchers have evaluated empirically the usefulness of various object-oriented design diagrams, e.g., (Hadar and Hazzan 2004; Dzidek et al. 2008; Bennett et al. 2008). Unfortunately, these evaluations focus mostly on class diagrams, or partial runtime views such as sequence diagrams, partly because runtime architectures have been difficult to obtain using previous technology.

More recent empirical evidence is paying greater attention to the importance of understanding the runtime structure of an application. (Lee et al. 2008) report on an empirical study where a participant expressed the need to understand "how objects connect to each other at runtime when I want to understand code that is unknown: an object diagram is more interesting than a class diagram, as it expresses more how [the system] functions".

Other opinions. Many experienced designers have recognized the importance of paying closer attention to the runtime structure of object-oriented applications. Trygve Reenskaug, the creator of the Model-View-Controller design pattern (Reenskaug 1979) and one of the earlier object-oriented methods (Reenskaug 1996), has been advocating an approach that makes explicit the following facts about code (Reenskaug 2008):

- What is the network of communicating objects?
- How are the objects interlinked?

- How do the objects interact?

Reenskaug advocates however a fundamentally different paradigm. On the other hand, SCHOLIA can help a developer gain a better understanding of the above questions, but for existing Java code bases and development methodologies, requiring only annotations.

8.1.1 Summary of previous work on design diagrams

Previous work recognized the importance of object diagrams, which show the runtime structure of a system, in addition to the value of class diagrams. Unfortunately, the previous tool support to extract meaningful object diagrams is still immature compared to the tools available for class diagrams.

SCHOLIA fills a previously neglected space, that of hierarchical static object diagrams. Hierarchy makes an object diagram scale effectively to show the object structures of an entire application, instead of just the interactions between a small set of objects. Moreover, we showed how a hierarchical object diagram can map intuitively onto a standard runtime architecture. Thus, SCHOLIA bridges even more closely object diagrams and descriptions of runtime architectural structure.

8.2 Architectural Description

Architectural description evolved independently from object-oriented design diagrams. Indeed, bridging and reconciling these two descriptions has been the subject of debate and research (Garlan et al. 2002b; Khammaci et al. 2005).

Code architecture vs. runtime architecture. There are many analogues between object-oriented design diagrams and architectural descriptions. For instance, the architectural analogue to a class diagram is a code architecture or module view. Similarly, the analogue of an object diagram is the *runtime architecture*.

Runtime architecture. Software architecture research recognized early on that a component in the runtime architecture of an object-oriented system would consist of objects and communication between them, such as procedure calls (Garlan and Shaw 1993; Shaw and Garlan 1996). In particular, such an architecture would not show inheritance relationships. (Garlan and Shaw 1993) state that “while inheritance is an important organizing principle for defining the types of objects in a system, it does not have a direct architectural function. In particular, in our view, an inheritance relationship is not a connector, since it does not define the interaction between components in a system”.

In SCHOLIA, a runtime architecture shows only objects, domains and relations between objects, and does not show inheritance relations. In contrast, some object models, e.g., those by (O’Callahan 2001), inspired from the Alloy object modeling notation (Jackson 2002), show objects, types as well as inheritance relations.

Relating runtime architecture to code. Unfortunately, several software architecture references are often imprecise when they relate a runtime architecture to object-oriented code (Garlan et al. 2002a). For instance, one of the standard books on software architecture, "Views and Beyond" by (Clements et al. 2003), suggests using a class diagram to represent a runtime architecture, then argues that "representing component instances as classes doesn't work when a component appears multiple times in a system" (Clements et al. 2003, p. 161).

Indeed, several approaches relate object-oriented modeling notations to architectural descriptions by mapping an architectural "component" to one or more *classes* or *packages* (Khammaci et al. 2005, Fig. 4) (Chardigny et al. 2008). In SCHOLIA, an object graph contains only runtime entities, i.e., objects and domains. And when analyzing conformance, SCHOLIA relates runtime component instances to runtime objects and their child objects, rather than static classes or packages.

Representation of runtime architecture. We designed SCHOLIA to work with a standard representation of a runtime architecture as a Component-and-Connector view (Shaw and Garlan 1996; Clements et al. 2003). There are alternate methods for modeling architectures, e.g., Fundamental Modeling Concepts (FMC), and their corresponding mappings between object-oriented code and architectural models (Tabeling and Gröne 2003).

8.2.1 Visualization of Software Architecture

Software visualization presents information in a way that takes into account the cognitive limitations of humans. Several software visualization techniques address the issues of diagram size or complexity. For instance, a hierarchical representation and the associated ability to expand or collapse elements has been shown to be effective for software architecture (Storey et al. 1999; Malton and Holt 2005). The RIGI visualization system (Müller and Klashinsky 1988) and its follow-up SHRIMP VIEWS (Storey et al. 1998) produce hierarchical views of the code architecture. Similarly, RELO (Sinha et al. 2006) shows hierarchical class diagrams. In RELO, the developer manually adds the classes of interest to each diagram and the tool lays them out. In other words, there is no automated static analysis behind the user interface.

SCHOLIA leverages the power of hierarchy, and represents a hierarchical object graph as a nested graph with domains (tiers) and objects (components) inside those domains. This allows expanding and collapsing objects or domains to achieve different levels of abstraction. In addition, in a C&C architecture, the architect can view the architecture at the top level, as well as drill into each component's sub-architecture.

One could argue that previous attempts to apply these architectural visualization techniques to the runtime structure of object-oriented systems have been lacking mainly in terms of the underlying program analyses they used previously, rather than shortcomings of their visualization techniques.

8.2.2 Summary of previous architectural description

The rich body of work on architectural description has long recognized the importance of documenting and reasoning about the runtime architecture of a system. SCHOLIA ascribes to the

same goals, and focuses on the runtime structure of object-oriented systems. As such, SCHOLIA benefits greatly from the large body of work on architectural description.

When reasoning about the runtime architecture of an object-oriented system at compile-time, the ideas and techniques of ownership types seem fundamental. First, ownership types provide a coarse structure of an application with a granularity larger than an object or a class, which previous approaches recognized as important (Sefika et al. 1996a). Second, ownership organizes a flat object graph into an ownership tree, and hierarchy provides abstraction and scalability by enabling both high-level understanding and detail. Third, different places in the hierarchy can distinguish between different instantiations of the same class that have distinct conceptual purposes and correspond to different elements in the design, which previous approaches identified as crucial to obtain meaningful object models (Lam and Rinard 2003). Fourth, the types can conservatively describe all possible aliasing that could take place at runtime, and information about aliasing is crucial for architectural analyses. Finally, ownership types can convey architectural intent, more so than a static analysis that computes aliasing information automatically without relying on annotations, as the negative result by (Rayside et al. 2005) demonstrates. So in the next section, we discuss ownership types which SCHOLIA leverages.

8.3 Ownership type systems

The SCHOLIA annotations implement the ownership domain type system (Aldrich and Chambers 2004), and the extensions from linear type systems in its AliasJava predecessor (Aldrich et al. 2002c). There are many ownership type systems (Clarke et al. 1998; Noble et al. 1998; Clarke 2001; Boyapati et al. 2003a; Aldrich and Chambers 2004; Dietl and Müller 2005; Potanin et al. 2006; Lu and Potter 2006; Schäfer and Poetzsch-Heffter 2007; Dietl et al. 2007; Müller and Rudich 2007), and new ones appear regularly.

We first discuss various expressiveness features in an ownership type system (Section 8.3.1), related type systems (Section 8.3.2), previous case studies evaluating ownership types (Section 8.3.3), and their inference (Section 8.3.4).

8.3.1 Expressiveness

Ownership type systems can be broadly characterized as *owner-as-dominator* or *owner-as-modifier*.

Owner-as-dominator. In an *owner-as-dominator* type system, any access to a child object must go through its owning object (Clarke et al. 1998; Noble et al. 1998). Such type systems are acknowledged to be too restrictive. As a result, they would not easily support annotating code after the fact. Also, because making an object owned by another object restricts access to the owned object, this forces more objects to be peers, and leads to clutter at the top level in the object graph. In addition, the ownership domain type system supports both this notion of strict encapsulation, as well as logical containment, which can make an object only conceptually part of another, without restricting access to the contained object.

Owner-as-modifier. An owner-as-modifier type system, e.g., (Müller and Poetzsch-Heffter 1999; Dietl and Müller 2005; Müller and Rudich 2007), supports strictly encapsulated objects[1], peer objects[2], and arbitrary `readonly` references, as long as only the owner can modify the object. Such a type system is fairly flexible. However using such annotations for architectural views is problematic because a separate analysis would have to resolve the `readonly` annotations in order to represent those objects in the object graph.

Other disciplines. There are other ownership disciplines. For instance, (Schäfer and Poetzsch-Heffter 2007) enforces a *boundary-as-dominator* property, and has the notion of a "loose domain", which is a form of an existential domain. Again, a separate analysis must resolve these domain annotations in order to soundly represent the corresponding objects in the extracted object graph.

Domain parameters. Some ownership type systems support ownership parameters. But others do not, e.g., (Dietl and Müller 2005). In object-oriented programming, it is typical to produce classes that are reused in different contexts. In particular, reusable or library code is often parametric with respect to the object ownership structure. For instance, a `List` object does not own its elements. Otherwise, those elements would not be accessible to the outside. As a result, the `List` class typically takes an ownership domain parameter for its elements. And every instance of that class must bind all the domain parameters on the class to other domains that are in scope. The object graph extraction then resolves these parameters, and ensures that the relevant object relationships appear in the global application architecture.

Domain parameters add to the annotation burden. However, as I was adding annotations to the subject systems (Chapter 4), I noted how adding these annotations can help identify tight coupling through unexpected domain parameters.

Generics. Existing ownership systems differ in their treatment of generics. Generic Ownership (Potanin et al. 2006) encodes generics in a strict owner-as-dominator model. Generic Universes (Dietl et al. 2007) encodes generics into an owner-as-modifier type system. We currently follow SafeJava (Boyapati 2004), and treat generics and ownership domains as orthogonal, perhaps at the cost of more verbose annotations. Adding existential domains may help make our annotations less verbose. For example, if an existential domain can correspond to a "raw type" in generics, then the annotation can be omitted in some cases.

Single vs. multiple contexts per object. Most ownership type systems support assume a single *context* per object (Clarke et al. 1998). As a result, the owner of an object is another object. Instead of having objects directly inside other objects, ownership domains use an extra level of hierarchy and group related objects inside a *domain*.

Simple Loose Ownership Domains (SLOD) (Schäfer and Poetzsch-Heffter 2007) hard-code the equivalent of one private and one public (or *boundary*) domain per object.

[1]The `rep` annotation is equivalent to using a private domain, e.g., `OWNED`.
[2]The `peer` annotation is equivalent to our `OWNER` annotation.

8.3.2 Related type systems

Related to ownership types are *confined types* and *region types*.

Confined types. Confined types enforce package-level confinement (Bokowski and Vitek 1999; Grothoff et al. 2001). They track that instances of a class are used within a given package. A *package* in confined types is roughly a package-level static ownership domain, and thus fairly coarse. As a result, confined types do not seem capable of assuring an instance-based runtime architecture. In particular, using confined types, one cannot distinguish between two instances of the same class that are used by different classes, within the same package. In addition, confined types do not have confined type parameters since all the packages are globally accessible. As a result, they ave a lower annotation overhead than ownership types.

Static class fields, which are really global variables, are challenging for most ownership type systems. However, confined type systems can deal readily with code that uses static variables. In addition, the low annotation overhead makes using confined types attractive, at least for highly unstructured code. In future work, it might be useful to leverage confined type annotations to extract an architectural view of a system.

Region types. Also related are region type systems (Boyapati et al. 2003b). Unlike a domain, which can represent any group of objects, a region represents a group of objects that are deallocated together. Region types do not protect access to the objects in a region; any object that can name a region can access the objects inside it. On the other hand, region types allow splitting an object across multiple regions. From an architectural standpoint, it may be beneficial to have that expressiveness. But it is also intuitive to treat a runtime object as an indivisible unit of computation and state.

Effects systems. Some effects systems, e.g., (Greenhouse and Boyland 1999), implement ownership-like systems that do not strongly encapsulate, so they may be somewhat similar to ownership domains. But effects systems require describing somewhat precisely the reading and writing of mutable state by a method. In contrast, ownership domain annotations require specifying only the domains of a method's formal parameters, the domain of a method's return value, and optionally the domain of a method's receiver. In many cases, formal method parameters are annotated with a fairly imprecise annotation such as `lent` to indicate temporary aliasing within the method's body.

8.3.3 Case studies for ownership types

Researchers of ownership types have not reported significant experience with most ownership type systems on real code. Many systems are paper-only designs (Lu and Potter 2006; Schäfer and Poetzsch-Heffter 2007). Only a few systems, notably Ownership Domains (Aldrich et al. 2002c; Aldrich and Chambers 2004), Universes (Dietl and Müller 2005) and Generic Ownership (Potanin et al. 2006), have been implemented (ArchJava 2003; Universes 2007; OGJ 2005), and even fewer systems have been evaluated in substantial case studies on

real object-oriented code (Aldrich et al. 2002c; Hächler 2005; Abi-Antoun et al. 2007a; Nägeli 2006). Many systems have been evaluated only to check if they can express the canonical iterator example. Others have applied ownership types to the standard design patterns in isolation. However, many expressiveness challenges arise in real object-oriented code, and when the same objects are involved in several design patterns at once. In addition, there are multiple ways to implement a standard design pattern.

(Hächler 2005) documented a case study in applying the Universes type system (Müller and Poetzsch-Heffter 1999; Dietl and Müller 2005) on an industrial software application and refactoring the code in the process. Although the subject system in the case study was relative large (around 55,000 lines of code), Hächler annotated only a portion of the system, and did not report the exact number of annotated lines of code. Hächler also manually generated visualizations of the ownership structure. In contrast, during my case studies, I used object graphs to visualize the ownership structure, and adjusted the annotations accordingly.

(Nägeli 2006) evaluated how the Universes and ownership domain type systems express the standard object-oriented design patterns (Gamma et al. 1994). However, in real world complex object-oriented code, design patterns rarely occur in isolation (Riehle 2000). My case studies indicated that it is often these subtle interactions, combined with the single ownership constraint of the type system, that can make adding the annotations difficult in some cases.

In the process of evaluating SCHOLIA, I conducted and reported on some of the largest case studies to date in applying ownership types to real object-oriented code.

8.3.4 Ownership inference

Ownership inference is a separate problem and an active area of ongoing research. Ownership inference algorithms use static analysis (Aldrich et al. 2002c; Agarwal and Stoller 2004; Cooper 2005), dynamic analysis (Werner and Müller 2007), or a mix of static and dynamic analysis (Wren 2003). A compile time inference, e.g., (Aldrich et al. 2002c), is preferable to a dynamic analysis, since the annotations have to soundly describe all possible ownership structures at runtime. However, many static analyses are unscalable, and it is precisely large systems that require annotation inference.

To my knowledge, no previous fully automated inference algorithm can create multiple domains in one object and meaningful domain parameters. In SCHOLIA, these are critical for representing the architectural intent, such as the separate UI and MODEL tiers in Aphyds (Chapter 2.2, Page 31).

Some ownership inference techniques adopt a restrictive notion of ownership (Ma and Foster 2007), infer only strictly encapsulated objects and unaliased objects, do not map their results back to a type system, do not infer domain parameters (Ma and Foster 2007; Liu and Milanova 2007), or infer imprecise long lists of domain parameters (Aldrich et al. 2002c).

I am optimistic that active research in this area, e.g., (Milanova 2008; Liu and Smith 2008), will significantly reduce the cost of adding the annotations, and thus, potentially benefit SCHOLIA's adoption.

8.3.5 Summary of previous work on ownership type systems

SCHOLIA builds on much research in ownership type systems, and uses one of the state-of-the-art ownership type systems. Most of the research in ownership types has focused on specifying and enforcing invariants in the code. To our knowledge, SCHOLIA is the first approach that uses a static analysis to leverage the ownership type annotations in a program, in order to reason about higher-level architectural representations of the code.

8.4 Static analysis of the runtime structure

SCHOLIA uses program analysis to leverage the ownership type annotations in the program. In this section, I discuss previous static analyses that extract static object diagrams or object graphs. We first discuss object graph analyses (Section 8.4.1), points-to analyses (Section 8.4.2) and then shape analyses (Section 8.4.3).

8.4.1 Object graph analyses

We distinguish static analyses that do not require annotations from those that do.

8.4.1.1 Annotation-free analyses

Several static analyses produce object graphs without requiring annotations, and produce non-hierarchical object graphs.

WOMBLE (Jackson and Waingold 2001) starts with a class diagram and uses heuristics for container classes and multiplicities to refine the object model. The follow-on tool, SUPER-WOMBLE (Waingold 2001), uses additional heuristics for merging types but does not attempt to be sound. The unsoundness is an engineering tradeoff that is claimed to produce correct object models in practice, by masking problems due to other weaknesses of the analysis (namely, that it is flow-insensitive). SUPERWOMBLE also uses built-in and user-defined abstraction rules for containers that coalesce a chain of edges in the object model into a single edge (Waingold 2001). SUPERWOMBLE also analyzes all classes that are transitively referenced (through constructor calls, field references, etc.) from the root set of classes. To avoid analyzing a large number of classes, most of which would not affect the output, a *stop-analysis configuration file* controls what classes or packages the tool analyzes (Waingold and Lee 2002).

AJAX^3(O'Callahan 2001) uses a sound alias analysis to build a refined object model as a conservative static approximation of the heap graph reachable from a given set of root objects. However, AJAX does not use ownership and produces flat object graphs. AJAX relies heavily on post-processing raw object graphs, such as by eliding all "lumps" with more than seven incoming edges or eliding all subclasses of a given type, e.g., `InputStream` (p. 248). Moreover, the object models that AJAX generates tend to expose internal implementation details (p. 252). SCHOLIA does not suffer from this problem since the annotations typically store an object's internal implementation details in private domains. On the other hand, AJAX is able to detect fields that are actually unused. In addition, AJAX can automatically and soundly split classes in the object model, i.e., determine that an object is indeed of type `Y` and not of type `Z`—even if `Z` is a subclass

of Y, and without any information other than the code. Finally, AJAX's heavyweight but precise alias analysis does not scale to large programs.

PANGAEA (Spiegel 2002) produces a flat object graph without an alias analysis and is unsound. The PANGAEA output for JHotDraw (Fig. 4.19) is even more complex than that of WOMBLE (Fig. 4.18). However, having the ability to display flat object graphs for programs that lack annotations can still be useful. Indeed, I ported the open source PANGAEA[4] tool to Eclipse, to display the object structure of an unannotated system, and perhaps assist a developer in the process of annotating an unfamiliar system—though a flat object graph is often unreadable.

(Rayside et al. 2005) proposed a static object graph analysis based on Rapid Type Analysis (RTA) (Bacon and Sweeney 1996), which produced unacceptable over-approximations for most non-trivial programs. In SCHOLIA, the ownership annotations prevent the static extraction analysis from merging objects too much or too little.

8.4.1.2 Annotation-based analyses

Lam and Rinard (Lam and Rinard 2003) proposed a type system and a static analysis (which I refer to here as LR) whereby developer-specified annotations guide the static abstraction of an object model by merging objects based on *tokens*. LR supports a fixed set of statically declared global tokens, and their analysis shows a graph indicating which objects appear in which tokens. Using token parameters, the same code element can be mapped to different design elements depending on context. Token parameters are similar to ownership parameters, which predated them by several years (Clarke et al. 1998), though the Lam and Rinard paper does not explicitly relate the two.

Unlike ownership domains, there is a statically fixed number of tokens, all of which are at the top level, so LR cannot show hierarchy such as a `listeners` object nested within a `Model` object (Fig. 2.3(a)). In contrast, the ownership domains within an object define a sub-architecture of contained objects, and in the case of recursive types, the domain structure is hierarchical and unbounded in depth.

The LR paper does not mention inheritance, and the LR formal system omits it (Lam and Rinard 2003, Fig. 10). LR has no proof of soundness either with or without inheritance. LR's only case study was an order of magnitude smaller than one of my larger case studies, e.g., JHotDraw (15 KLOC vs. 1.7KLOC). If I were to apply LR to JHotDraw anyway, ignoring inheritance, LR would show at least 200 objects in the top-level tokens. In contrast, SCHOLIA applies abstraction by ownership hierarchy and by types to show an order of magnitude fewer objects in the top-level domains.

[3] AJAX (O'Callahan 2001) is not publicly available. O'Callahan was kind enough to send me the sources for the tool. But I was unable to run AJAX successfully, even on trivial examples because it requires a specific, obsolete environment, and has various undocumented dependencies. This explains why the flat object graphs that I show for comparison with SCHOLIA are the output of WOMBLE (Jackson and Waingold 2001) or PANGAEA (Spiegel 2002), which I was able to obtain and run.

[4] PANGAEA is publicly available at: `http://page.mi.fu-berlin.de/spiegel/pangaea/`

8.4.2 Points-to analysis

Points-to analysis is a fundamental static analysis to determine the set of objects whose addresses may be stored in variables or fields of objects (Andersen 1994). The research literature on points-to analysis goes back several decades, and I do not claim to summarize it here. I discuss briefly several points-to analyses for Java. They can broadly organized along the following dimensions:

- *Context- sensitive vs. insensitive*: an analysis is context-insensitive if it analyzes a method m only once, combining all the calling states of $obj_1.m(\overline{x_1})$ and $obj_2.m(\overline{x_2})$. A context-sensitive version will distinguish between these two call sites. As a result, a context-sensitive analysis can be less scalable than a context-insensitive one. SCHOLIA's analysis does not distinguish between calling contexts, but distinguishes objects based on their domains. So, one can consider domains as a form of context-sensitivity.
- *Flow- sensitive vs. insensitive*: an analysis is flow-sensitive if the order of the statements in a program affects the result of the analysis. SCHOLIA's analysis is flow-insensitive and does not consider the program's control flow.
- *Object- sensitive vs. insensitive*: in an object-insensitive analysis, a field f declared in a class C has a class-level scope ($C::f$). This allows a points-to analysis to distinguish between fields that belong to two different classes, e.g., $C::f$ vs. $D::f$. However, an object-insensitive analysis cannot distinguish between fields that are declared in a given class, but belong to different instances of that class, e.g., $obj1.f$ vs. $obj2.f$, where $obj1$ and $obj2$ are field declarations of type C. In some cases, the ability to distinguish between locations that belong to different objects improves substantially the precision of analysis (Milanova et al. 2005). On the downside, object-sensitivity makes an analysis unscalable.

All previous points-to analyses produce non-hierarchical graphs (Tonella and Potrich 2004; Milanova et al. 2005). The SCHOLIA static analysis can be considered flow-insensitive and object-insensitive but domain-sensitive (Refer to discussion in Section 3.6.3, Page 118), and produces hierarchical object graphs.

An Object Flow Graph (OFG) (Tonella and Potrich 2004) is similar to an object-sensitive points-to graph. It tracks the lifetime of objects from their creation point to their assignment to program variables.

Object Process Graphs (Eisenbarth et al. 2002) use points-to analysis to statically recover all possible execution traces for a given object. One application of these graphs is protocol validation.

Soundness. Not all points-to analysis have soundness proofs. Some of them are written as pseudo-code, instead of transfer functions, which makes it difficult to compare between these different analyses.

Analysis results. Although points-to analysis results are often used for compiler optimization, the value of points-to analysis for program understanding was previously identified (Tonella et al. 1997; Tonella and Potrich 2004). However, the result of the analysis is typically used only intra-procedurally, since a points-to graph for an entire system would probably be unreadable.

In the same vein as SCHOLIA, (Milanova et al. 2002) uses the results of a points-to analysis to construct an Object Relation Diagram, which is a class diagram where the type of the pointed-

to object is potentially more precise than the declared type. To our knowledge, SCHOLIA is the first approach to abstract the output of a static points-to analysis into a hierarchical runtime architecture represented as a C&C view, which is then used to analyze conformance to a target runtime architecture.

Points-to summary. Most points-to analyses also abstract all the objects that could be created at one allocation site into one node in the points-to graph. As a result, most points-to analyses achieve a granularity that is no coarser than an object or a set of objects. In SCHOLIA, a node in a hierarchical object graph includes all the objects at an allocation site within a domain[5], together with all the objects collapsed underneath that object, based on the ownership domain parameters, as well as the type structures, when using abstraction by types. In addition, a domain is a conceptual groups of objects, and provides a granularity coarser than that of an individual object with its collapsed substructure.

8.4.3 Shape analysis

Our analysis creates a graph that summarizes possible relationships among objects at runtime. Shape analysis, e.g., (Sagiv et al. 1999), is related, but differs on several counts. First, shape analyses have not been demonstrated to scale to programs with more than a few thousands of lines of code. Second, shape analysis represents objects that are being used by the program using unique materialized objects, while it summarizes objects that are not in use. In contrast, our analysis, once it merges two objects in a domain, never separates them. So, shape analysis could produce more precise results for small non-hierarchical graphs. But our analysis can separate two objects that are in distinct domains, because the underlying type system guarantees they can never alias. Finally, shape analysis produces very precise shape graphs consisting of nodes to represent a set of objects, and edges to represent points-to relations. However, a shape graph is non-hierarchical in the sense that all the nodes in a graph are at the same level, and objects are not collapsed underneath other objects.

(Calcagno et al. 2009) proposed a shape analysis that can show, for a given method, the input and the output shape graph. This analysis works only *intra*-procedurally, which keeps the shape graphs manageable. However, if one were to apply the analysis to the whole program *inter*-procedurally, it is likely to produce very large graphs that would not convey any meaningful architectural abstraction.

Finally, a heavyweight shape analysis may achieve more precision than SCHOLIA in many cases. Although SCHOLIA sacrifices some precision to gain scalability of the analysis, it conveys architectural abstraction primarily through hierarchy.

8.4.4 Summary of previous static analysis of the runtime structure

While these approaches produce diagrams that are very useful, they typically extract design diagrams rather than architectural diagrams, in the sense that they convey little architectural

[5]Disclaimer: the object graph may not reflect `unique` objects which may be returned by a factory method. As discussed earlier, this requires a flow analysis to resolve the `unique` annotations.

abstraction (Section 1.4, Page 9). In particular, all previous static object graph analyses, points-to analyses, and shape analyses, including some that use annotations, extract non-hierarchical object graphs. Flat objects graphs do not scale, because the number of top-level objects in the object graph increases with the program size. More importantly, a flat object graph often does not provide sufficient architectural abstraction to enable analyzing conformance.

8.5 Dynamic analysis of the runtime structure

When dealing with runtime structure, many have intuitively preferred dynamic analysis. In this section, I discuss previous dynamic analyses that extract low-level diagrams. I first discuss general visualization-oriented approaches (Section 8.5.1), then approaches that make use of ownership (Section 8.5.2), then approaches that mix both dynamic and static analysis (Section 8.5.3).

8.5.1 Visualization of object structures

Many dynamic analyses focus on visualizing the object structures of a running system (De Pauw et al. 1993, 1994; Lange and Nakamura 1995; Sefika et al. 1996a; Koskimies and Mössenböck 1996; Jerding et al. 1997; Walker et al. 1998; Richner and Ducasse 1999; De Pauw and Sevitsky 1999; Gargiulo and Mancoridis 2001; Souder et al. 2001; Smith and Munro 2002; Oechsle and Schmitt 2002; De Pauw et al. 2002; Salah and Mancoridis 2004; Pacione et al. 2004; Reiss and Renieris 2005).

These dynamic analyses handle programs for which source code is not available, do not require source code annotations, and allow more fine-grained user interaction in producing a visualization. These task-focused views explain detailed interactions, help developers understand a program, or find low-level defects, such as memory leaks (De Pauw and Sevitsky 1999; Rayside and Mendel 2007). The extracted views have the granularity of individual objects and classes.

Many of these approaches extract one or more collaboration diagrams (Gschwind and Oberleitner 2003; Koskimies and Mössenböck 1996; De Pauw et al. 1994; Richner and Ducasse 1999; Walker et al. 1998), rather than a global object diagram for the entire system. A collaboration diagram that contains all objects and all invocations between them may be unusable, for anything but the smallest of systems. Most approaches allow the developer using the tool to focus the interaction diagram to include only specific method invocations, issued from a starting method of interest.

An alternative solution is to analyze an incomplete system, by including only classes of interest. SCHOLIA supports analyzing a portion of a system, and allows the use of "virtual fields" to soundly summarize the un-annotated portions.

In some cases, the recovered views highlight design patterns (Kramer and Prechelt 1996; Schauer and Keller 1998), but often, they are not architectural, because they are neither abstract nor global.

8.5.2 Dynamic ownership analyses

More closely related to SCHOLIA are dynamic analyses that infer the ownership structure of a running program based on its heap structure (Hill et al. 2002; Rayside et al. 2006; Flanagan and Freund 2006; Mitchell 2006).

In general, dynamic analyses have the advantages of being more scalable and more precise than their static counterparts. In addition, dynamic ownership analyses do not require a programmer to annotate her code with ownership type annotations. However, previous such analyses assume a strict owner-as-dominator model which cannot represent many design idioms. In such a model, a higher-level object cannot collapse underneath it not many low-level objects, so they end up cluttering the top-level diagram.

Table 8.1 has a comparison of dynamic ownership analyses and static object graph analyses, some of which require neither annotations nor the source code (they operate on the bytecode version of the program).

Table 8.1: Comparison of dynamic ownership analyses and static object graph analyses.

	Ownership	Scalable Analysis/Viz. (Kind)	Design Intent	Sound Analysis	Comments
Dynamic ownership analyses					
Rayside et al. (Rayside et al. 2006)	Inferred	Yes/Yes (Matrix)	No	Yes	
Mitchell (Mitchell 2006)	Inferred	Yes/No (Flat)	No	Yes	
AARDVARK (Flanagan and Freund 2006)	Inferred	No/No (Flat)	No	Yes	
DINO (Hill et al. 2002; Noble 2002)	Inferred	Yes/Yes (Both)	No	Yes	
Potanin et al. (Potanin et al. 2004)	Inferred	Yes/Yes (Both)	No	Yes	
PTIDEJ (Guéhéneuc 2004)	None	No/No (Flat)	Some	No	
Static object graph analyses					
AJAX (O'Callahan 2001)	None	No/No (Flat)	No	Yes	Bytecode
WOMBLE (Jackson and Waingold 2001)	None	No/No (Flat)	Some	No	Bytecode
PANGAEA (Spiegel 2002)	None	Yes/No (Flat)	No	No	Source
Lam and Rinard (Lam and Rinard 2003)	Tokens	Yes/No (Flat)	Yes	Partial	Source
SCHOLIA	Annotated	Yes/Yes (Hierarchy)	Yes	Yes	Source

Hill, Noble and Potter (Hill et al. 2002) and (Potanin et al. 2004) used dynamic analyses and showed both matrix and graph views of ownership structures and demonstrated that ownership is effective at organizing runtime objects. Several others followed suit (Mitchell 2006; Rayside et al. 2006; Flanagan and Freund 2006).

(Rayside et al. 2006) characterize sharing and ownership and produce a matrix display of the ownership structure. They later used the results of this analysis to investigate memory leaks (Rayside and Mendel 2007).

Similarly, (Mitchell 2006) uses lightweight ownership inference to examine a single heap snapshot rather the entire program execution, and scales the approach to large programs through extensive graph transformation and summarization.

(Flanagan and Freund 2006) propose a dynamic analysis with a 10X-50X overhead to reconstruct each intermediate heap from a log of object allocations and field writes. Then, they apply a sequence of abstraction-based operations to each heap, and combine the results into a single object model that conservatively approximates all the observed heaps. Their tool, AARDVARK, has the notion of ownership and containment and uses simple heuristics to choose the most appropriate generalization. In addition, AARDVARK's dynamic object diagrams have multiplicities, which SCHOLIA's static object diagrams do not have.

This body of work showed that ownership does provide abstraction, and is effective at organizing large object graphs. SCHOLIA uses the same key insight but in a static analysis which must address several additional challenges. A static analysis for object-oriented programs must also deal with issues of aliasing, recursion, inheritance, soundness, precision and scalability.

Dynamic ownership analyses are descriptive and show the ownership structure in a single run of a program. In contrast, the Ownership Object Graph that SCHOLIA obtains at compile time is prescriptive and shows ownership relations that will be invariant over all program runs. Thus, this dissertation proposes a new class of object graphs that is new, important, and valuable.

8.5.3 Mix of static and dynamic analysis

PTIDEJ (Guéhéneuc 2004) uses a dynamic analysis to refine a class diagram obtained using a static analysis, but with manual input. For example, PTIDEJ was evaluated on JHotDraw, and the UML class diagram it produced did not fit on one page.

(Tonella and Potrich 2002) combine static and dynamic analysis to extract object diagrams from a C++ library, as well as interaction diagrams (Tonella and Potrich 2003).

8.5.4 Summary of previous dynamic analysis of the runtime structure

The analyses we discussed in this section obtain useful, low-level diagrams, that have the granularity of individual objects or classes. Some dynamic object diagrams also used hierarchy effectively. However, by definition, a dynamic object diagram cannot show all possible objects and relations.

SCHOLIA is the first approach that uses ownership types to add hierarchy to a *static* object diagram. In SCHOLIA, a hierarchical object graph conveys architectural abstraction, as we discussed in Section 1.4 (Page 9), and can be abstracted into a standard runtime architecture. The next section focuses on previous work that extracts architectural diagrams.

8.6 Architectural extraction

There is much previous work in the area of architectural extraction, which reverse-engineers high-level architectural views of a system. Architectural extraction is also known as *architectural recovery*, *architectural reconstruction*, *reverse architecting* or *architectural discovery* (Koschke 2008; Ducasse and Pollet 2009).

Many of the previous techniques deal only with the code architecture, rather than the runtime architecture, which is the subject of this dissertation. Unfortunately, many papers confound the runtime and the code architectures. They either do not explicitly classify an architectural view they extract as either a code or a runtime architecture, or use the term "component" to really mean a "package", "module" or a collection of classes (Tvedt et al. 2002). This observation is corroborated by (Ducasse and Pollet 2009) in their extensive survey of previous architecture extraction techniques: "Because it is complex to extract architectural components from source code, those are often simply mapped to packages or files. Even if this practice is understandable, we think it limits and overloads the term component" (p. 587).

In the following discussion, I restate some of the contributions of previous work using a terminology that is consistent with the rest of this document, and clarify whether the end result is a code architecture or a runtime architecture.

Most architectural extraction follows the *extract-abstract-present* strategy (Krikhaar 1997). It first *extracts* some information from the code, the *source model*, *abstracts* that source model into a *high-level model*, then *presents* it, either visually, or using an architectural description language.

8.6.1 Extracting a source model

An architectural extractor can use static analysis, dynamic analysis, or a mix of the two to extract a source model.

8.6.1.1 Static extractors

Many static extractors extract information such as package structure, class structure, dependencies such as inheritance and method calls (Murphy and Notkin 1995). Some of that information, such as a directory or package structure, is naturally hierarchical. Other approaches require the hierarchical containment information as a separate input. For instance, the Software Bookshelf (Finnigan et al. 1997), of which (PBS 2000) is an instantiation, has the notions of:

- hasParts/isPartOf, e.g., a System might have constituent SubSystems as parts;
- contains/isContainedIn.

In PBS, a human specifies the containment information separately from the facts that a tool extracts from the source code. For example, PBS uses two decomposition files which represent decomposition information for a software system. The first is an "established decomposition" that a human supplies or verifies, and the second is an "adopted decomposition" that the toolkit guesses.

(Mendonça and Kramer 2001) developed an approach and a tool, X-RAY, to extract the runtime architecture using only static analysis, but from procedural C code. X-RAY combines com-

ponent module classification, syntactic pattern matching, and structural reachability analysis. It is unclear that the approach can handle object-oriented code.

8.6.1.2 Dynamic extractors

A dynamic extractor monitors a system's execution, obtains snapshots of the runtime heaps, and analyzes the snapshots either online or offline (Walker et al. 1998).

8.6.1.3 Mixed extractors

Some extractors combine both static and dynamic analysis (Richner and Ducasse 1999).

8.6.1.4 Summary of previous work in extracting source models

With the exception of object graph analysis, points-to analysis and shape analysis, which I discussed in Section 8.4 (Page 283), most static extractors do not track objects precisely. Instead, they represent their structural information with respect to files, directories[6], packages or classes, rather than objects. For example, they express that a class is part of some package, or a package is nested inside some other package.

In SCHOLIA, ownership type annotations provide the containment information. Moreover, SCHOLIA uses object-level, i.e., an object is "part of" another object. This enables SCHOLIA to distinguish between different instances of the same class that are in different domains, as well as between instances of the same class in the same domain but with different actual domain parameters.

8.6.2 Abstracting a source model into a high-level model

Abstraction techniques *abstract* a source model into a high-level architectural view. (Hochstein and Lindvall 2005) survey various of these approaches. Following (Sartipi and Kontogiannis 2004) and others, we broadly organize abstraction techniques into *clustering* and *pattern matching* methods.

8.6.2.1 Clustering

Clustering identifies higher-level architectural entities by gradually grouping lower-level entities. Many architectural extraction approaches use clustering to decompose a system into a collection of hierarchical subsystems, allowing nodes to be collapsed or expanded (Tzerpos and Holt 1996). For example, RIGI (Müller et al. 1993), and several of the following tools such as DALI (Kazman and Carrière 1999), which was superseded by ARMIN (Kazman et al. 2002), and SHIMBA (Systä et al. 2000), use this technique. Some tools, e.g. ARMIN, have a scripting interface so a developer can write scripts to aggregate information and produce higher-level views.

[6] In Java, the package and the directory structures must mirror each other. Languages such as C++ and C# allow the package and the directory structures to be different.

Clustering approaches rely on naming conventions (Kazman and Carrière 1999), directory structures (Richner and Ducasse 1999), or graph clustering algorithms (Sartipi and Kontogiannis 2003a; Maqbool and Babri 2007).

Several tools simply apply the same notions of clustering for procedural code to object-oriented systems. E.g., BUNCH (Mancoridis et al. 1999) organizes Java functions into modules, by clustering entities in a module dependency graph.

Clustering techniques are attractive because they often produce acceptable results, are scalable and can be mostly automated. However, clustering techniques rarely recover a meaningful decomposition because they do not go beyond the structural relationships explicitly declared in the code. For example, architectural extraction studies that used graph clustering algorithms reported that software developers often used trial and error with the clustering parameters (Kazman and Carrière 1999; Christl et al. 2005). As a result, clustering approaches are increasingly incorporating user input at the detriment of their automation.

Clustering methods can be complementary to SCHOLIA and may help with annotating an unfamiliar system. For instance, two strongly connected clusters may suggest creating two top-level domains corresponding to the two clusters. A small cluster that interacts with almost all others may indicate `shared` objects.

8.6.2.2 Pattern matching

Broadly speaking, pattern matching techniques map low-level elements to higher-level elements by searching for patterns. (Ducasse and Pollet 2009) provide a good overview of previous approaches that extract architecture based on pattern matching. Examples include extracting components according to queries over a relational database containing the code (Kazman et al. 2002), identifying architectural actions via event sequences in run-time execution that match a state machine (Schmerl et al. 2006), or using a user-provided map to relate source code entities to architectural components, as in Reflexion Models (Murphy et al. 2001).

One of the pattern-matching approaches, DISCOTECT (Schmerl et al. 2006), deserves additional discussion because it instruments a running system and extracts a built C&C runtime architecture that is rich with architectural styles and types. In place of annotations, DISCOTECT requires mapping events from a runtime trace to architectural counterparts, e.g., a method invocation leads to the creation of a port. In addition, it may be possible to reuse a mapping across several similar systems, which is not the case with ownership type annotations. Because DISCOTECT is a dynamic analysis, the results reflect only the particular inputs and exercised use cases. Also, DISCOTECT generates non-hierarchical C&C views that show one component for each instance created at runtime. Another analysis must post-process DISCOTECT's output to consolidate similar component instances. Finally, DISCOTECT only extracts the built architecture of a system and does not analyze its conformance to a target architecture.

At the lowest level, some approaches try to detect the standard design patterns (Keller et al. 1999). Many design patterns are more micro-architectural, low-level and local than the global architectural structure that SCHOLIA can extract or analyze. In SCHOLIA, the annotations typically encode the decomposition of a system into high-level patterns, such as Model-View-Controller.

Pattern matching techniques tend to recover a handful of the standard design patterns, and suffer from a large number of false positives.

Pattern-matching techniques tend to be less scalable than clustering-based methods. But no pattern-matching approach can guarantee that every class in the system will be assigned to a subsystem.

None of them has the desired precision. The resulting decompositions are either not meaningful to a software engineer, or they cover only pieces of the whole system.

8.6.2.3 Summary of previous work in abstracting source models

Most approaches mostly automate extracting a source model, but require the developer to guide the abstraction step. SCHOLIA is unusual in that developer-specified annotations also guide the extraction. The annotations help the analysis that extracts the source model to distinguish between objects that are in different domains. This helps achieve additional precision and bring the source model closer to an architect's mental model of the runtime architecture.

In SCHOLIA, an OOG groups objects based on the architectural intent captured by the annotations, and on the object ownership and type structures, not according to where objects were declared in the program or some naming convention. The abstraction by types during the OOG extraction involves additional manual input, and requires a developer to specify trivial types or design intent types.

In SCHOLIA, once the annotations are added to the program to capture the architectural intent, they can evolve with the program.

8.6.3 Case studies in architectural extraction

There are several published case studies in architectural extraction.

8.6.3.1 Non-object-oriented systems

Most published architectural extraction case studies studied big legacy systems written in procedural languages, rather than object-oriented code, which is the subject of this dissertation. However, the processes these case studies followed are quite instructive and we followed similar ones during our own case studies.

For example, a successful case study extracted the code architecture of a 30-KLOC C system (Harris et al. 1995) and a multi-MLOC C system (Linux) (Bowman et al. 1999).

To determine the architectural structure of a legacy system, (Tzerpos and Holt 1996) used a "hybrid" process that combines facts extracted from the code and information derived from interviewing developers. These steps include: collecting "back of the envelope" designs from project personnel; extracting raw facts from the source code; collecting naming conventions for files; clustering code artifacts based on naming conventions; creating tentative structural diagrams, and collecting the reactions of the developers to these tentative diagrams; and so on, until they converged to a code architecture. They concluded that there is a reasonably well-defined sequence of steps to go through to extract a code architecture. Indeed, the steps we undertook while evaluating SCHOLIA during the field study were somewhat similar, although we dealt with the runtime architecture, and did not use clustering.

The Apache modeling project (Gröne et al. 2008) used FMC and manually extracted the architecture of Apache, written in C. The architectural extraction seems to have involved ad-hoc manual techniques and many people—many students enrolled in a class. The only tool used for the analysis of the source code transformed the C source code into a set of syntax highlighted and hyperlinked HTML files (Gröne et al. 2002). The authors justify not using more advanced tools by saying that "an important amount of information needed for the conceptual architecture is not existent in the code and therefore cannot be extracted by a tool" (Gröne et al. 2002).

8.6.3.2 Object-oriented systems

There a few published case studies in the architectural extraction of object-oriented systems.

Several case studies have studied the Jigsaw system, which has 300 classes (Chardigny et al. 2008; Medvidovic and Jakobac 2006), to evaluate their architectural extraction techniques. However, they focused on the code architecture.

(Medvidovic and Jakobac 2006) point out that available tools are often unable to discover a relationship that is implemented indirectly, e.g., by using instances of container classes, such as `Vector`, `Map`, `List`, to store objects of some other application class. This is further complicated by introducing user-defined container classes. SCHOLIA can readily address those cases. For example, in the Listeners system (Chapter 2), the class `Model` does not directly declare a field reference of type `BarChart`. Rather, the class `Model` declares a `List` of `Listener` objects (Fig. 2.4, Page 36). After edge lifting, SCHOLIA can show a points-to relation between a `Model` object and a `BarChart` object (Fig. 2.3(b)).

Many architectural extraction studies use various sources of information extrinsic to the code, with no clear exit criteria. It is also fairly common for different people to extract very different architectures.

In SCHOLIA, the annotations are not completely arbitrary, and have to typecheck. During the LbGrid field study (Section 4.8, Page 161), we only added annotations, typechecked them, and occasionally discussed a snapshot with a developer. The measurable success criteria are to minimize the number of objects in the top-level domains, and reduce the number of remaining annotation warnings.

8.6.3.3 Evaluating an extracted architecture

Evaluating the quality of a extracted architecture is subject to debate, with no generally accepted evaluation criteria. More generally, this appears to be a common issue in the empirical evaluation of reverse engineering tools. (Tonella et al. 2007) state that "the same piece of information recovered from the code may be immensely useful or completely unusable depending on the end user who is performing the current software engineering task and depending on the amount of knowledge the user already has about the system". Indeed, this was one of the challenges that this dissertation work faced.

One approach to measure the "goodness" of an extracted architecture is to compute various structural metrics. Indeed, clustering methods often use this approach to evaluate the quality of the result. For example, a clustering is "good" if the clusters are reasonably sized and exhibit low coupling and high cohesion (Systä et al. 2000).

Another common way to evaluate an extracted architecture is to compare it against a target or reference architecture. This is the approach taken by SCHOLIA and many others.

8.6.3.4 Summary of previous case studies in architectural extraction

The scale of the systems we analyzed using SCHOLIA may pale in comparison to previous case studies that analyzed the code architecture of large systems. However, SCHOLIA is a type-based technique that requires developers to specify architectural intent using annotations. This makes using SCHOLIA to analyze a system with millions of lines of code (MLOC) prohibitively costly, without annotation inference. Despite this limitation, SCHOLIA is the first entirely static approach that can extract a runtime architecture. For comparison, the closest prior work that used annotations to extract object models (Lam and Rinard 2003) was evaluated on one 1700-line system.

8.6.4 Summary of previous work in architectural extraction

Previous architectural extraction focused predominantly on the code architecture. The previous work that addressed the runtime architecture intuitively preferred using dynamic analyses. (Schmerl et al. 2006) state that "determining the actual runtime architectural configuration of a system using static analysis is, in general, undecidable". (Ducasse and Pollet 2009) affirm that "static information is often insufficient for [software architecture reconstruction] since it only provides a limited insight into the runtime nature of the analyzed software; to understand behavioral system properties, dynamic information is more relevant" (p. 580).

To my knowledge, no previous approach extracts statically a runtime architecture for an object-oriented system. The closest to that would be the runtime structure extracted by a static analysis, which I discussed in Section 8.4 above. Most flat object graphs are too low-level to be considered architectural views. SCHOLIA is the first approach to demonstrate that the static extraction of runtime architectures from object-oriented code is indeed feasible.

8.7 Architectural synchronization

SCHOLIA requires the ability to compare the built and the designed architecture. Several techniques and tools have been proposed for differencing and merging architectural or design views.

Landmark-based algorithms. We group several algorithms that have been proposed for differencing hierarchical information under the category of "landmark-based algorithms": they have been proposed in the context of program differencing, e.g., JDiff (Apiwattanapong et al. 2004), Dex (Raghavan et al. 2004), and design differencing, e.g., UMLDiff (Xing and Stroulia 2005). These algorithms are based on the assumption that the entities they are trying to match are uniquely named and many nodes match exactly. This enables them to recognize the unchanged nodes first and use them as "landmarks" to efficiently identify the other changes. However, these algorithms are unable to match nodes based on structure alone or based on structure and highly non-unique semantic information, such as entity types. For instance, a heuristic solution with a

worst-case $O(N^3)$ supporting arbitrary move, copy and glue operations was tested on instances where more than 80% of the nodes matched exactly (Chawathe and Garcia-Molina 1997). As a result, these algorithms are less suitable for comparing architectural views, as they will perform poorly when all the nodes are renamed, or when most of the renamed nodes are concentrated in one area of the tree such as when entire subtrees are renamed. This may be atypical when comparing two versions of a given program or a design model at a given level of abstraction. In our architectural views, most names are transient or automatically generated. Both THP and MDIR would still work even in the total absence of semantic information, i.e., using tree structure only. For instance, in the Aphyds and Duke's Bank examples, our inputs had more than half of their nodes renamed. Finally, none of these algorithms offer the ability to manually force or prevent matches. It may be possible to easily add the ability to prevent matches to some of them (e.g., JDiff), but adding the ability to force matches could be substantially more complicated.

Tree alignment vs. tree edit. Tree differences can be represented using tree alignment instead of tree edit distance. Each alignment of trees actually corresponds to a restricted tree edit in which all the insertions precede all the deletions. Algorithms based on tree alignment can detect unbounded deletes and can generalize to more than two trees, something not easily done with tree edit distance algorithms (Jiang et al. 1994). But the memory requirements of tree alignment algorithms, for the tree sizes and branching factors that are typical of our inputs, would be several orders of magnitude higher than those of MDIR—$O(2^{2d}N^2)$, where d is the maximum degree of the tree.

Graph matching approaches. Exhaustive graph matching algorithms, based on variants of the A* algorithm (Messmer 1996), do not scale beyond a few dozen nodes (Hlaoui and Wang 2002). In the context of architectural views, Sartipi proposed an approach for architectural extraction using a variant of the A* graph matching algorithm, but with an optimization that may cause it to miss the optimal solution in some cases (Sartipi and Kontogiannis 2003b).

More scalable, heuristic-based approaches, such as spectral methods, perform poorly when the graphs are not nearly isomorphic. Furthermore, these algorithms occasionally miss the optimal solution (Conte et al. 2004). Others, such as the Similarity Flooding Algorithm (SFA), have an accuracy of around 50% (Melnik et al. 2002). The accuracy of MDIR is above 90% on a roughly similar range of graph sizes. Furthermore, SFA relies heavily on labels, which are different when the graphs originate from different domains, even if they express the same relationships: "while matching of an XML schema against another XML schema delivers usable results, matching of a relational schema against an XML schema fails" (Melnik et al. 2002).

(Mandelin et al. 2006) proposed probabilistic matching based on label, region, type or position information, but the approach requires training the *evidencers*. Mandelin et al. also mention that a simple greedy search algorithm does not work in many cases.

Model transformation. Graph transformation approaches, surveyed by Mens and van Gorp (Mens and Van Gorp 2005), tackle the same problem, but use a different set of assumptions. First, in many graph grammars, productions do not delete vertices and edges, which effectively

prohibits insertions and deletions, one of our requirements. Second, graph transformation approaches do not attempt to find the optimal transformation that would preserve properties of view elements. Finally, most graph transformation approaches do not yet offer the same level of automation as the tools illustrated in Section 5.5 (Page 192).

Consistency management. There is significant work in the area of viewpoints, view merging and inconsistency management, e.g., (Easterbrook and Nuseibeh 1996; Egyed 2006). 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 a built and a 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.

Much of the work in view consistency analyzes the consistency of different but related views, typically at the same level of abstraction, such as a UML class diagram and a UML sequence diagram. This is a problem in *horizontal conformance* (Ducasse and Pollet 2009). On the other hand, this dissertation is about analyzing consistency between views at different levels of abstraction, namely an implementation and a target architecture, which is a problem in *vertical conformance* (Ducasse and Pollet 2009).

8.8 Built-in conformance

Analyzing conformance, after the fact, between an architecture and an implementation is a fundamentally difficult problem. So several approaches attempt to avoid the problem by using code generation, style-guidelines, library-based solutions and language-based solutions.

8.8.1 Code generation

Because of the difficulty of maintaining and extracting high-level models of a system, some approaches make the high-level model the primary asset and achieve conformance by generating an implementation from an architecture (Shaw et al. 1995; Moriconi et al. 1995). However, code generation guarantees only initial conformance. To maintain conformance, developers must refrain from changing the code directly. Instead, they must always change the models then regenerate the implementation from the updated models. Such an approach may work in certain domains, but is often not adoptable for general purpose applications, because a developer is no longer free to edit the code directly. This restricts architects and developers from working at the appropriate level of abstraction. More importantly, such an approach does not handle legacy code. SCHOLIA can analyze the conformance of a system after the fact, handles legacy systems, and requires mostly adding annotations, without re-engineering the system to a different language.

8.8.2 Style guidelines

In order to maintain conformance, some approaches require an implementation to follow strict style guidelines that prohibit sharing mutable data between components (Luckham and Vera 1995). The problem with style guidelines is that there are usually no tools to check them.

8.8.3 Library-based solutions

Library-based solutions achieve conformance by requiring developers to implement theiwr system on designated architectural frameworks or middleware. For instance, adopters of the C2 ADL can use a specific framework to implement their design (Medvidovic et al. 1996). Similarly, FRACTAL (Bruneton et al. 2006) defines multiple levels of conformance, and supporting higher levels requires implementing additional interfaces.

In addition to forcing developers to use specific frameworks, such approaches often require developers to follow strict guidelines to avoid introducing architectural violations. There are no tools to check that an implementation obeys those rules (N. Medvidovic, personal communication, 2008).

8.8.4 Language-based solutions

ArchJava (Aldrich et al. 2002b) was the first language-based solution that could guarantee, at compile time, communication integrity between object-oriented code and the intended runtime structure. ArchJava, however, expresses architecture through Java language extensions, and requires re-engineering a system to respect ArchJava's type system (Aldrich et al. 2002a; Abi-Antoun and Coelho 2005; Abi-Antoun et al. 2007a). We discussed in detail the relation between SCHOLIA and ArchJava in Section 7.5.5 (Page 240)

8.8.5 Summary of previous work in built-in conformance

Previous approaches to enforce built-in conformance have several drawbacks. They require developers to use code generation, mandate specialized architectural middleware or frameworks, or impose strict style guidelines without providing tools to enforce those guidelines. Others require radical language changes that incorporate architectural constructs at the expense of severe implementation restrictions. While these approaches may be adoptable in certain restricted domains, they often do not address existing systems. In contrast, SCHOLIA supports analyzing the conformance, after-the fact, of a system written in a general purpose, widely-used object-oriented language (Java), and that uses available frameworks and libraries.

8.9 Architectural conformance

Others have recognized the drawbacks of enforcing built-in conformance, and worked on approaches to analyze conformance between an implementation and an architectural view after the

fact. In fact, many architectural extraction approaches are designed with the goal of analyzing conformance (Ducasse and Pollet 2009)[7]. In this section, we focus only on approaches that analyze horizontal conformance between the implementation and the architecture.

We classify the previous work in terms of approaches that address the code architecture (Section 8.9.1), and others that address the runtime architecture (Section 8.9.2).

8.9.1 Conformance analysis of the code architecture

One of the earliest and most influential techniques for analyzing conformance to a code architecture is Reflexion Models (RM) (Murphy et al. 2001). Although RM works on the code architecture only, I modeled SCHOLIA closely after RM and discussed their differences and similarities in detail in Section 6.6.4 (Page 218).

(Knodel and Popescu 2007) performed a comparative analysis of Reflexion Models and two other conformance analyses techniques for the code architecture and indicated they have similar expressiveness.

Many approaches use variations on Reflexion Models. For instance, (Fiutem and Antoniol 1998) check if a system's code architecture represented as a class diagram conforms to a design specified in the OMT modeling language, a precursor to UML. A tool translates both the C++ source code and the target architecture into an intermediate representation, compares the two, and identifies added or removed classes, attributes, operations, association and inheritance relations. The comparison uses a maximum match algorithm that computes the best mapping between source code classes and entities of the class design based on string edit distance. Because the entities in the two representations are at the same level of abstraction, there is no need to map one representation to the other, as in Reflexion Models. Their experiments on several C++ systems ranging 5K to 50K confirm that assuming exactly matching names for classes gave poor results of design-code traceability. That work, among others, justifies SCHOLIA's use of a structural comparison that can detect renames, to compare the built and the target architectures.

Similarly, the approach by (Guo et al. 1999) works with object-oriented code such as C++. But the end result shows C++ files as the leaf nodes, and summarizes the conformance between call relations and variable access relations.

(Postma 2003) describes a method for verifying a module architecture using relational partition algebra (RPA), which targets high-level architectural rules only. Then they check the conformance of an extracted view against a target view.

(Eichberg et al. 2008) use annotations to define "ensemble" of packages, classes, etc. These ensembles define a module view that is orthogonal to the code structure. In principle, an ensemble could consist of fields, though the approach does not address the issue of possible aliasing. In addition, the annotations describe in the code the desired structure of the system, and check it continuously as part of the build process.

They have the notion of "part of" at the level of files, classes or packages, but not at the level of objects. Unlike SCHOLIA, their annotations are not typecheckable. Finally, the SCHOLIA annotations do not describe the desired structure of the system in the code.

[7](Ducasse and Pollet 2009) list 34 approaches to extract architecture, and indicate that 12 of them are used to analyze conformance (Table 1, p. 576). Of these, 9 deal with *vertical conformance* (Table 5, p. 584).

(Kontogiannis et al. 1995) check structural compliance using the notion of concept-to-code mapping. In this approach, a concept language models abstract properties of a desired code fragment. The pattern matching process is based on the Markov model, and a similarity measure between an abstract pattern and a piece of code is defined in terms of the probability that the abstract pattern can generate that piece of code. To reduce the complexity of the required computations, they use dynamic programming.

8.9.2 Conformance analysis of the runtime architecture

Previous work in analyzing conformance to a runtime architecture use either dynamic analysis (Section 8.9.2.1) or static analysis (Section 8.9.2.2).

8.9.2.1 Dynamic analysis

Intuitively, many have preferred using dynamic analysis to analyze conformance to a runtime architecture, by monitoring a few program runs of the system (Sefika et al. 1996b; Madhav 1996). In contrast, SCHOLIA uses only static analyses, and thus, can make claims about all possible executions.

(Madhav 1996) instruments an Ada program to produce events at runtime that are tested for conformance against a reference architecture documented in the RAPIDE ADL.

PATTERNLINT (Sefika et al. 1996b) combines static and dynamic visualization and analyzes conformance by displaying various complementary views such as a "data sharing graph", and inter-class call relations. The rules are converted to Prolog.

(Turner et al. 2003) extend UML object diagrams into Visual Constraint Diagrams and check at runtime that a UML object diagram satisfies constraints. These constraint are over instances of classes and express conditions that should not occur if the object-oriented program is correct, e.g., that a linked list must not contain a cycle. (Crane and Dingel 2003) do something similar, but require developers to use the Alloy object modeling notation.

(Shomrat and Yehudai 2002) showed that using AspectJ (Kiczales et al. 1997) to enforce architectural restrictions is not an ideal choice. Although design problems are cross-cutting, they often concern static events or structural properties that cannot be captured by existing pointcut languages. Static analysis, which we use in our approach, seems better suited to ensure structural properties.

8.9.2.2 Static analysis

Some approaches use static analysis and purport to handle the runtime architecture of object-oriented systems. However, (van Dijk et al. 2005; Díaz-Pace and Campo 2005; Blech et al. 2006) map a *component instance* in a runtime architecture to a *class* in object-oriented code. Such a mapping is more suitable for the *code architecture*. Implicitly, they make the assumption that there is a single instance of each class in the system.

In SCHOLIA, runtime component instances are not classes. Rather, components correspond to objects, i.e., instances of classes. For example, a framework, when instantiated, may contribute one or more component instances to an architecture. This justifies the use of *object diagrams*

instead of class diagrams as the closer analogy for a runtime architecture (Clements et al. 2003, p. 103).

Bauhaus (Raza et al. 2006) is a static analysis toolkit that supports a points-to analysis, etc. Bauhaus lets a user specify the high-level module view (or a hypothesis thereof) and map the concrete modules onto the architecture. Then Bauhaus compares a high-level module view to the concrete modules and their dependencies using a similar technique to Reflexion Models. Presumably, Bauhaus can handle Java code. However, the two published case studies used two large-scale and complex applications, namely, the C compiler sdcc and the GNU C compiler gcc, with 100 and 500 KLOC, respectively (Koschke and Simon 2003).

8.9.3 Case studies in architectural conformance

Several case studies evaluated the conformance analysis using case studies. When dealing with the code architecture, the source model can be obtained relatively easily. This allows a technique such as Reflexion Models (RM) to scale to large code bases. For instance, (Murphy and Notkin 1997) analyzed a 1.2-MLOC system written in C.

(Rosik et al. 2008) conducted an *in vivo* study using a variant of Reflexion Models.

8.9.4 Conformance measurement

There are several possible measures of architectural violations in source code.

(Sarkar et al. 2006) measure how many back-calls or up-calls an implementation violates with respect to the layers in a target code architecture.

(Laguë et al. 1998) compute metrics that compare the layers in a designed and a built code architecture.

SCHOLIA is complementary, focuses on the runtime architecture, and relates a designed and a built runtime architecture.

8.9.5 Summary of previous work in architectural conformance

There is much previous work in analyzing conformance to a code architecture. However, an approach designed for the code architecture would not work on the runtime architecture. This is because one code entity can map to multiple components in a runtime architecture, and similarly, multiple code elements could correspond to the same object at runtime.

To our knowledge, no previous work can analyze the conformance of a runtime architecture, and statically relate the runtime component instances in a target architecture to runtime objects. Still, SCHOLIA is similar in spirit to previous work such as Reflexion Models and many similar variants. However, SCHOLIA differs from that work on two counts. First, SCHOLIA uses a more sophisticated source abstraction method and extracts a richer source model that reflects the application's hierarchical runtime structure, instead of its code structure. Second, SCHOLIA relates the source model to a high-level model using a more powerful structural comparison. The structural comparison does not rely on unique identifiers, and compares two hierarchical architectural views after the fact. Finally, SCHOLIA deals with hierarchical source models, high-level models, and maps.

8.10 Traceability

Traceability has long been recognized as important (Lindvall and Sandahl 1996; Spanoudakis and Zisman 2005), and is strongly related to conformance. Similarly to *horizontal* and *vertical conformance*, (Lindvall and Sandahl 1996; Spanoudakis and Zisman 2005) discuss *horizontal* and *vertical traceability*. Despite the plethora of approaches to achieve traceability, effective tool support remains a challenge (Spanoudakis and Zisman 2005; Oliveto et al. 2007), and is of limited use in industrial settings.

General purposes approaches use various information retrieval techniques to recover traceability links between use cases, between design diagrams and code classes, or between test cases and code classes (Giulio et al. 2000; Antoniol et al. 2002; De Lucia et al. 2007). A tool based on such a technique produces measures of similarity (De Lucia et al. 2008).

ARCHEVOL (Nistor et al. 2005) maintains traceability links between a code architecture and the implementation, once a human provides the tool with an initial mapping between the architecture and the implementation.

Previous tools that do establish traceability to the code, often do so with respect to a code architecture, i.e., they relate some artifact in a high-level model to a *class* or a *package* in the code structure, rather than to objects in the application's runtime structure. In contrast, SCHOLIA can establish traceability between the built runtime structure and an intended runtime architecture. To our knowledge, SCHOLIA is the first approach that allows a developer to trace from a component, a connector or a port in runtime architecture, obtained entirely statically, to the corresponding object references in a general purpose object-oriented language like Java. This facility was previously available only when tracing from UML class diagrams to Java code.

8.11 Summary of related work

SCHOLIA fills an important gap in extracting statically a hierarchical runtime architecture from a general purpose object-oriented language like Java, and enforcing communication integrity against a target architecture.

Chapter 9

Discussion and Conclusion

In this chapter, I revisit the requirements on a proposed solution and discuss how well SCHOLIA meets them (Section 9.1). I then discuss some of SCHOLIA's limitations (Section 9.2), its usefulness and usability (Section 9.3), possible future work (Section 9.4), and finally conclude.

9.1 Satisfaction of the SCHOLIA requirements

SCHOLIA meets many of the requirements from Section 1.8 (Page 22).

9.1.1 Overall Approach

RQ O1 – Hierarchical architectural models: In SCHOLIA, both the designed and the built architectures are hierarchical;

RQ O2 – Static analysis: The SCHOLIA object graph extraction uses a static analysis. The analyses that abstract an object graph and analyze communication integrity are also static;

RQ O3 – Arbitrary implementation code: I evaluated SCHOLIA successfully on existing object-oriented code that used available libraries;

RQ O4 – After the fact analysis: I evaluated SCHOLIA on existing code that others had developed. In many cases, the systems had no documented architectures.

RQ O5 – Automation: I automated many parts of SCHOLIA, as I discuss below. The part of SCHOLIA that would benefit from significantly more automation is the process of adding the annotations to a program.

9.1.2 Annotations

RQ ANN1 – Language support for annotations: I designed SCHOLIA's annotations to use existing language support for annotations for most of the cases. There are a few cases that the existing annotation standard (Bloch 2004) cannot handle. So, we currently use brittle block comments for those. However, upcoming versions of Java are likely to adopt the JSR 308 proposal (Ernst and Coward 2006), which allows annotations in more places such as on generic type arguments;

RQ ANN2 – Real object-oriented code: I was able to add annotations to several real object-oriented systems, which used inheritance, recursion, etc.;

RQ ANN3 – Expressiveness: The annotations that I added to existing object-oriented code implement the ownership domain type system and typecheck for the most part. In the process of adding annotations, I identified expressiveness challenges in the type system that must be addressed in future work;

RQ ANN4 – Automation: I implemented a tool to insert default annotations (Section A.4.4, Page 326), and another tool to typecheck the annotations.

9.1.3 Architectural Extraction

RQ EXT1 – Summarization:

RQ EXT2 – Hierarchy: The ArchRecJ tool extracts hierarchical object graphs.

RQ EXT3 – Object soundness: We proved formally that an extracted object graph has exactly a unique representative for each runtime object (Section 3.3, Page 88).

RQ EXT4 – Edge soundness: We proved formally that an extracted object graph has edges that correspond to all possible runtime points-to relations between the representatives of the runtime objects (Section 3.3, Page 88).

RQ EXT5 – Traceability: The ArchRecJ tool allows tracing from each node or edge in an extracted object graph, including from a lifted edge, to the underlying lines of code.

RQ EXT6 – Precision:

RQ EXT7 – Scalability: The object graph extraction static analysis, even though it is a whole program analysis, does seem to scale. In particular, it avoids known scalability bottlenecks such as object-sensitivity.

RQ EXT8 – Automation: I developed a tool, ArchRecJ, to extract an object graph from an annotated program, with a good response time. The tool assists a developer with selecting the input to the abstraction by types, and refining the object graph interactively.

9.1.4 Architectural Comparison

RQ COMP1 – No unique identifiers: The structural comparison does not assume that the architectural view elements have unique or persistent identifiers.

RQ COMP2 – No ordering: The structural comparison does not assume that an architectural view has an inherent ordering among its elements.

RQ COMP3 – Insertions, deletions, and renames: The structural comparison does detect elements that are inserted, deleted and renamed across two architectural views.

RQ COMP4 – Hierarchical moves: The structural comparison does detect elements moved up or down a number of levels in the hierarchy.

RQ COMP5 – Manual overrides: The structural comparison allows a user to force or prevent matches between selected view elements. The comparison does then take these constraints into account to improve the overall match.

RQ COMP6 – Type information optional: The structural comparison does not assume that the view elements have type information that matches exactly. The empirical evaluation showed that the comparison can recover a correct mapping from structure alone if

necessary, or from structure and type information if type information is available. The comparison also takes advantage of any available type information, and avoids matching elements that have incompatible types.

RQ COMP7 – Disconnected and stateless operation: The structural comparison works after the fact, in a disconnected and stateless mode. It does not rely on the ability to monitor or record any structural changes to an architecture.

RQ COMP8 – Automation: The ArchSynchro tool (Section 5.4.2, Page 189) can synchronize two architectural C&C views.

9.1.5 Architectural Conformance

RQ CHK1 – Communication integrity: Extracting a sound object graph is a prerequisite for enforcing communication integrity. Indeed, the extracted object graph implied by the ownership annotations must show all objects and all possible communication between those objects. The object graph abstraction and conformance analysis preserve soundness since they may only add but not subtract edges, e.g., in the form of lifted edges in the built C&C view or summary edges in the conformance view. However, we do not present a soundness proof that relates a Runtime Object Graph (ROG) to a conformance view.

RQ CHK2 – Few false positives: The evaluation showed that if the built and the designed architectures have a similar hierarchical decomposition and a similar number of components at each hierarchy level, the conformance analysis does not produce too many false positives. Moreover, a developer can intervene at several steps in the approach to reduce the number of false positives, by fine-tuning the annotations, controlling the abstraction step, guiding the structural comparison, etc.

RQ CHK3 – Traceability: I developed a tool, CodeTraceJ, to allow the developer to trace from each convergent or divergent component or connector in a conformance view, including a summary connector, to the underlying lines of code;

RQ CHK4 – Automation: I developed a tool, ArchCog, to abstract an object graph into a built runtime architecture. I also developed a tool, ArchConf, to compare the built architecture to a target architecture, analyze communication integrity in the target architecture, and display a conformance view.

9.2 Limitations

SCHOLIA suffers from several limitations.

9.2.1 Overall Approach

Semi-automation. SCHOLIA is not a push-button approach. Architects and developers have to provide many of the abstractions and manually interpret the results. This is both a strength and a weakness. It is a strength because it enables SCHOLIA to obtain meaningful abstractions, in contrast to a fully automated approach which is more likely to infer a high-level model that may not match the architect's mental model (Wong et al. 1995; Murphy et al. 2001).

One benefit of extracting an architecture based on annotations is that the abstraction is not hard-coded in the tool. Indeed, "many tools only support showing a previously abstracted view [. . .] Maintainers might understand the software better through abstractions they created themselves, rather than through the prefabricated 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" (Storey et al. 1999).

On the other hand, the degree of manual input in applying the SCHOLIA may not be worth the effort for systems that are not business-critical, and may preclude its immediate practical adoption. Currently, most of the effort required to use SCHOLIA is in manually adding the annotations to a program.

Based on our field study results, SCHOLIA currently requires roughly a person-week of effort for a 30-KLOC system. For many systems, this cost is high, and could be s a significant barrier to industrial adoption, and thus to practitioners achieving the approach's benefits.

Batch-oriented interaction. In SCHOLIA, a developer iterates the process of adding annotations which control the abstraction by ownership hierarchy, then the object graph extraction which controls the abstraction by types. She then abstracts the object graph by selecting various options, and structurally compares the abstracted object graph to a target architecture. Based on the comparison results, she refines the annotations until the extracted object graph has a similar hierarchical decomposition and shows a similar number of components as the designed architecture. Finally, she must investigate whether the reported divergences or absences are true architectural violations or could be addressed by refining the annotations, and iterating the process one more time.

Overall, the process of refining the extracted architecture seems somewhat awkward. The architect must notice and analyze architectural anomalies, assume some of them are due to an incorrect ownership relationship in the source code, change the ownership annotations consistently to reflect the corrected ownership relationship, and then regenerate the architecture. Having to run a sequence of analyses and tools may make using SCHOLIA tedious and time-consuming.

9.2.2 Annotations

The annotations suffer from the following limitations.

Expressiveness challenges. Like any type system, the ownership domain type system has some expressiveness challenges. During our evaluation, we encountered several expressiveness challenges (Section 4.6.1.3, Page 135). In fact, most of our annotated programs still have annotation warnings remaining in them. One way to address these warnings would be to refactor the code. But having to refactor existing code to annotate it adds to the adoption cost of the approach. Ideally, we should extend the type system. We believe some of these expressiveness issues may be resolved in the type system by incorporating a few well-understood constructs that others have added to other ownership type systems, such as existential ownership (Clarke 2001; Krishnaswami and Aldrich 2005; Lu and Potter 2006). Other expressiveness limitations, such as dealing with static fields, may be harder to overcome using an ownership type system.

One potentially promising approach would be to use a type system which combines ownership types and confined types, such as (Potanin 2007).

Single ownership. The ownership domain type system used by SCHOLIA supports only single ownership, i.e., an object cannot be part of more than one ownership hierarchy. For instance, if an object is both a mediator in the Mediator pattern and a view in the Model-View-Controller pattern, it cannot be in two `MEDIATOR` and `VIEW` ownership domains at once. Proposals for *multiple ownership* lift this restriction in other type systems (Cameron et al. 2007).

Lack of ownership transfer. The ownership domain type system does not support *ownership transfer* either, i.e., an object's owner does not change —only `unique` objects can flow between any two domains. Some recent type systems lift this restriction and support ownership transfer (Müller and Rudich 2007).

Annotation inference. The main drawback of SCHOLIA seems to be the abundance of ownership annotations that are needed. The manual annotation effort is a potential obstacle for practical adoption, but ownership annotations are amenable to automated ownership inference, which could alleviate this problem, at least partially. With precise and scalable ownership inference, SCHOLIA can scale to large systems. Ownership inference is a separate problem and an active area of ongoing research.

Previous ownership inference techniques can infer encapsulated objects in private domains and unaliased objects (Liu and Milanova 2007; Ma and Foster 2007; Milanova 2008). But they do not infer public domains, do not infer domain parameters (Liu and Milanova 2007) or infer too many domain parameters (Aldrich et al. 2002c).

9.2.3 Architectural Extraction

SCHOLIA's architectural extraction suffers from the following limitations.

Abstraction by types. The ownership annotations, which control the abstraction by ownership hierarchy, are the main input to extract object graphs. The object graph static analysis also takes optional input to further merge objects based on their declared types. This additional input may be needed to reduce the number of objects at a given level of the hierarchy, and obtain a built architecture that is comparable to the designed one. But, unlike the ownership annotations which can be mechanically typechecked for consistency with each other and with the code, there is no way to automatically validate the types that a developer selects for the abstraction by types. As a result, selecting the trivial types or the design intent types may require some trial and error. However, that optional input cannot make an extracted object graph unsound.

Potential unsoundness. For soundness, an OOG requires a complete set of annotations. In particular, an OOG may be missing objects or edges if the external libraries used by the program create architecturally important objects or edges, but are incompletely annotated, or if the manually-specified virtual field annotations that summarize those libraries are unsound.

Handling dynamic reconfiguration. SCHOLIA does not currently capture dynamic architectural reconfiguration (Magee and Kramer 1996); it shows only the footprint of any such reconfiguration in the object graph. Moreover, SCHOLIA currently uses an ADL that describes the static architecture of a system but one that offers no facilities for specifying runtime architectural changes (Oreizy et al. 1998). Enriching the extraction analysis to describe possible dynamic configuration will also require using an ADL that can represent some architectural dynamism.

Handling distributed systems. SCHOLIA currently applies to applications that run in a single virtual machine, so it handles neither heterogeneous nor distributed systems (Magee et al. 1995; Mendonça and Kramer 2001).

Precision. SCHOLIA currently relies only on the aliasing precision that the annotations provides. Namely, that two objects in different domains can never alias. But two objects of compatible types, in the same domain, may alias. In the absence of more precise aliasing information, this can lead to a precision loss in some cases. To compensate for this limitation, a developer can specify more fine-grained domains, but, of course, this adds to the annotation burden. Ideally, a domain-aware alias analysis might be able to achieve the best of both worlds: take into account developer-specified annotations, achieve better precision and remain scalable.

Plain Old Java Objects (POJOs). I designed SCHOLIA for systems where each object is a Plain Old Java Object (POJO). SCHOLIA does not have any special handling for the parts of a system that use a component framework such as Enterprise Java Beans (EJB), aspect-oriented programming (Kiczales et al. 1997), etc. While SCHOLIA is a general purpose solution, it is possible that a domain-specific approach could achieve better results, or require less effort for certain classes of systems.

9.2.4 Architectural Comparison

Scalability. SCHOLIA uses structural comparison to compare the designed and the built architectures. If the views are very different, an automated structural comparison may fail to match the built and the designed views. In that case, the comparison will not produce useful results since all components will be absences (the comparison will delete all the elements from one view and add them to the other). The algorithm does allow the developer to manually match some view elements, but at the cost of additional effort. Finally, the algorithm is quadratic in the view sizes. So, while the algorithm scales to up to a few thousand nodes (Chapter 5), the comparison of very large architectures may be intractable.

9.2.5 Architectural Conformance

Architectural abstraction. Currently, in SCHOLIA, abstracting an object graph into a built runtime architecture requires interaction through a user interface, for example to soundly summarize private domains. Future work may specify abstraction rules that a tool can apply automatically to abstract an object graph into a C&C view. In addition, merging objects only based on

their ownership or type structures, while sufficient most of the time, is not fully general. Future work may define more general abstraction rules. For instance, a rule can map an entire domain to a component, or merge objects based on a predicate that takes into account the names or the types of those objects.

Architectural behavior. SCHOLIA currently supports analyzing the conformance of architectural structure and not of architectural behavior (Allen and Garlan 1994).

Object multiplicities. Currently, the object graphs extracted by SCHOLIA lack information about multiplicities.

9.3 Usefulness and Usability

In this section, we discuss SCHOLIA's usefulness and usability.

9.3.1 Usefulness

(Ducasse and Pollet 2009) classify the output of software architecture extraction as one of the following:

- **Architectural visualization:** i.e., a high-level view of the system organization;
- **Architectural description:** i.e., a description in an architectural description language (ADL);
- **Conformance analysis:** i.e., an extracted architecture enables analyzing conformance;
- **Architectural analysis:** i.e., an extracted architecture enables a quantitative or qualitative architectural-level analysis.

Throughout this dissertation, I concretely demonstrated how SCHOLIA provides value in each of the above areas.

Architectural visualization. In Chapter 4, I indicated several instances of how an extracted object graph highlights facts about the global program structure that may not be obvious from looking at the code.

Our evaluation does *not* claim to demonstrate that a visualization based on hierarchical object graphs can provide actual assistance to a third-party developer in completing a code modification task. Admittedly, properly evaluating such a claim requires a user study. In such a study, one could provide some developers with a class diagram, others with the code, with or without the ownership annotations, and the rest with an OOG. Then, one could measure if the developers who have access to the OOG can complete some code modification tasks faster or better than the ones who have access only to a class diagram or to the code. Such a study, however, is outside the scope of this dissertation.

Architectural description. In Chapters 6, 7, I discussed and evaluated how SCHOLIA can represent an extracted architecture as a C&C view in an ADL, which allows reusing much of the existing research in architectural modeling and analysis.

Conformance analysis. In Chapters 6, 7, I concretely demonstrated that an extracted architecture enables analyzing communication integrity in a target architecture.

A user study is not the only way of demonstrating value. For instance, using the CryptoDB case study in Section 7.8, (Page 250), I demonstrated how SCHOLIA can potentially be useful for threat modeling, by ensuring that the security architecture used in a security review shows all possible entry points and communication in the implementation.

Architectural analysis. I concretely demonstrated how one can enforce various global constraints on the architectural structural using the CryptoDB case study.

Future work. SCHOLIA's evaluation to date does not quantify the benefits of the approach in terms of finding errors or improving the ability to add new functionality to existing code. This leaves open several questions that future work might try to answer, such as: can an extracted OOG provide assistance to a developer performing a code modification task? Can an extracted OOG be useful to an architecture review board (ARB) during an architectural review (Maranzano et al. 2005)?

9.3.2 Usability

We briefly discuss the *usability* or ease of learning and applying the SCHOLIA approach. While we leave to future work a more formal usability evaluation by outside developers, we offer the following, more qualitative data.

Effort to apply. One measure of usability is the effort needed to apply the approach, and in particular, the annotation effort. Based on the 35 hours to annotate a 30-KLOC system, an experienced developer should budget about 1 hour per 1,000 lines of code when adding the annotations manually.

Effort to learn. Another measure of usability is that we were able to teach the approach during a half-day tutorial at the SEI SATURN professional event. The architects, researchers and experienced developers in attendance learned the approach and used the various tools in less than three hours. We provided the tutorial participants with a partially annotated CryptoDB system, and they were able to successfully run the tools on that system. The reader can refer to the tutorial handout (Abi-Antoun and Aldrich 2009c) for more information about the tutorial's contents and hands-on exercises.

When trying to teach the approach in an academic setting, however, we noticed that novice Java developers seemed to have difficulty grasping the annotation semantics and syntax.

9.4 Future Work

There are several avenues of future work that could be worth exploring.

9.4.1 Overall Approach

Demonstrate scalability. I evaluated the end-to-end SCHOLIA approach on several extended examples totaling around 40 KLOC. I also conducted a single field study on a 30-KLOC system to evaluate the object graph extraction. While these sizes may seem small, the static analysis of the runtime architecture is not yet mature compared to the analysis of the code architecture. For instance, the most relevant previous work was evaluated on a single 1,700-LOC system (Lam and Rinard 2003). Still, I would like to apply SCHOLIA to larger systems. For instance, extracting the architecture of Eclipse, which is currently over a million lines of code, would be a stretch goal. A more intermediate goal is to scale the approach to handle systems an order of magnitude larger than the ones we have used so far.

Quantify the benefits of runtime architectures. The relation between runtime architectures and design and coding tasks remains poorly understood. Even empirical studies that looked at various design diagrams focused on partial runtime views, such as sequence diagrams. It would be interesting to investigate the use of runtime architectures for various code modification tasks, to demonstrate concretely and quantitatively their benefits.

Relate runtime and code architectures. During my field study, I observed that experienced developers often structure their code architecture carefully. For instance, they place classes that serve different conceptual purposes into different packages, modules, or layers. They also define marker interfaces that do not define any methods, to indicate some design intent. But due to its static nature, a code architecture cannot represent the dynamic architecture of a system. My analysis is also influenced by many of these code attributes, such as marker interfaces. I would like to leverage how a code architecture is structured to display a runtime architecture, e.g., by overlaying layers in a code architecture and tiers in a runtime architecture. I would also like to use the extracted architecture to identify potential refactoring opportunities. For example, excessive merging in an extracted object graph may be due to a type structure that includes many classes which inherit from a constant interface, a practice which is considered an anti-pattern (Bloch 2001, Item #17).

Explore a more incremental, interactive approach. SCHOLIA currently does not provide instant gratification. In particular, a developer adding the annotations may need some of the knowledge provided by the extracted object graph, which makes the process highly iterative. So there is ample room to make SCHOLIA more interactive. For instance, better tools could help with refining the annotations based on visualizing the extracted object graph, or support more flexible ways to abstract an object graph into a component-and-connector architecture.

Evaluate usefulness and usability. In future work, it may be informative to have outside developers use the tools to independently evaluate their usefulness and usability.

9.4.2 Annotations

It may be helpful to improve the annotations and the tool support for adding them.

More flexible type system. One important area of future work on the annotations would be to extend the type system, to eliminate the remaining annotation warnings.

Non-ownership annotations. Ownership types have been around for over a decade (Clarke et al. 1998). However, they have yet to be adopted on a wide scale, perhaps due to the overhead of adding them to existing code bases. Perhaps simpler, non-ownership annotations might make an annotation-based approach more adoptable by practitioners.

Automation. Automating the process of adding the annotations can greatly help the adoptability of SCHOLIA by practitioners.

9.4.3 Architectural Extraction

Notational issues. SCHOLIA requires learning new techniques and displays the object graph in a notation that is different from widely adopted notations such as the Unified Modeling Language (UML). As a result, developers may not be interested in investing time to learn it. Grundy and Hosking mentioned that most dynamic visualizations bear little or no relation to static architecture visualization (design) notations, making them harder to understand and interpret (Grundy and Hosking 2003).

Layout issues. In architectural diagrams, color, size and width often convey specific meanings. Similarly, the location in a hierarchy is important, e.g., whether some object is above or below another (Koning et al. 2002). It is common to show an "EventBus" connector as wider or thinner than other components in the diagram. In some architectural styles, location matters. In the C2 architectural style, each component has a single top port and a single bottom port, notifications flow down, and requests flow up (Taylor et al. 1996). Without additional annotations, an OOG will use the same size, color for all objects. For instance, an OOG will display an `EventBus` component with the same size as a `Course` component, which may not be as informative. To partially alleviate the problem, one could define additional annotations to encode visualization attributes directly in code, as in the UML Graph approach (Spinellis 2003; Fowler 2004).

Add more precision. The focus of this research to date has been on soundness. As a next step, it may be useful to achieve better precision, e.g., by showing cardinality on object relations. One idea would be to use a heavyweight shape analysis on demand, to gain additional precision when displaying the object structures within a domain, by also leveraging the annotations that are already in place.

Support distributed architectures. SCHOLIA currently only works for applications that run on a single virtual machine. With the increasing popularity of architectural patterns such as service-oriented architectures, software systems are increasingly distributed. I plan to extend SCHOLIA to handle the runtime architecture of distributed systems.

Support multiple views. I want to explore how to produce multiple but consistent views of a runtime architecture of the same software system, e.g., one to show data flow and another to show control flow. SCHOLIA supports analyzing modules of an entire system. However, once you extract multiple runtime architectures for different modules of a system, it is unclear how to tie them together into an overall architecture.

Support architectural dynamism. The static analysis I developed is flow-insensitive and context-sensitive, which enables it to be scalable. As a result, the extracted architecture captures only the footprint of any dynamic architectural reconfiguration. With systems becoming increasingly dynamic, it may be useful to provide more precise information about possible architectural reconfiguration.

So far, I have mostly used entirely static analyses. Some extensions may require combining static and dynamic analysis, to achieve additional precision, track the dynamic loading of code as used by many modern plugin architectures, or account for the use of reflection or calls to native code.

9.4.4 Architectural Comparison

Splitting/Merging. In future work, it may be useful to enhance the structural comparison to detect the splitting or merging of components across two views.

9.4.5 Architectural Conformance

Continuous checking. It may be useful to make the conformance analysis more continuous, similar to continuous unit testing (Saff and Ernst 2005). This way, a developer can realize that she is violating the target architecture as soon she makes a code change with undesired architectural ramifications.

9.5 Conclusion and Broader Impact

As early as 1968, Dijkstra pointed out the importance of partitioning and structuring a system carefully, in addition to programming it correctly. Dijkstra put forth the notion of a layered structure, where one layer could only communicate with adjoining layers. The costs of adopting this organization for conceptual integrity would be offset by the gains in development and maintenance ease (Dijkstra 1968). Since then, there has been much work in formalizing the notion of software architecture. One promise was that specifying a software architecture in an architecture description language would enable various architectural-level analyses for performance, security and reliability.

Much of that promise has gone unfulfilled until now, partly for two reasons. First, all systems have an architecture, but very often, it is not explicitly documented. Second, the relationship between a designed architecture and the actual system implementation, including the built architecture, is unclear. The effectiveness of architectural analyses to improve software dependability in practice requires an implementation to correctly realize the carefully thought-out architecture.

Current object-oriented systems are slowly becoming the legacy systems of the future. 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 software evolution.

Previous attempts to relate the architecture to the implementation required developing programs on specific implementation frameworks, or specifying the architecture directly in code. Such proposals imposed strict implementation restrictions or non-backward-compatible language extensions. Indeed, re-engineering existing Java implementations to a research language that specifies the architecture within the code would be prohibitively expensive for the millions of lines of existing code that power our information age.

Today, practicing software engineers still face big challenges in understanding the global structure of a software system well enough to effectively evolve it, integrate it with other systems, or analyze the impact of a change. As qualities such as performance, reliability and security become more critical, it is increasingly important for engineers to understand not just the code structure, but also the run-time structure of a system. Since many software systems exceed a million lines of code in size, architects must rely on architectural documentation to achieve this understanding—yet this documentation is often missing or out of date, and must extracted from code.

Statically extracting runtime architectures from code had been an open problem. However, reasoning accurately about qualities like reliability and security cannot consider only the typical case, and requires understanding all possible communication between components, which suggests that a *sound* approach based on static program analysis is ideal. Moreover, in today's object-oriented systems, the runtime structure showing objects and their relations is often quite different from the decomposition of the static code structure into source files, classes and packages.

This dissertation addresses the problem for existing object-oriented languages and existing designs, requiring only annotations, using the SCHOLIA approach. SCHOLIA is the first entirely static approach that guarantees, at compile time, communication integrity between code in a widely used object-oriented language and a rich, hierarchical description of an architect's intended runtime architecture.

SCHOLIA models runtime architectures as a hierarchy of objects, with architecturally significant objects near the top of the hierarchy and data structures demoted further down. Because architectural hierarchy is not readily observable in a program written in a general purpose programming language, SCHOLIA uses ownership annotations in the program to impose local information about object encapsulation and logical containment.

This dissertation demonstrated the feasibility of sound, static extraction and conformance analysis of the runtime architecture of object-oriented systems. An evaluation on several real systems showed that SCHOLIA can establish traceability between an implementation and an intended runtime architecture, and identify interesting structural differences. Admittedly, the

approach is costly—requiring roughly a person-week of effort for a 30 KLOC system, because the approach relies on manually adding type-like annotations ubiquitously throughout the source code to specify architectural intent that is missing in a general purpose programming language. This cost is a significant barrier to widespread industrial adoption.

However, even with its current cost, the cost-benefit of SCHOLIA may still be favorable, for many business-critical systems. Furthermore, as the field study demonstrated, it is both valuable and possible to add the annotations, extract object graphs and analyze conformance of only a core sub-system of a larger system.

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

Enabling the extraction of sound runtime architectures can make a major impact on the ability of engineers to understand and effectively evolve complex software systems. Practitioners can now trace between the architecture and the code. They can also use the traceability information to determine what part of a system to change, or where performance or security problems are likely to arise. Easy access to trustworthy architectural diagrams thus could eventually facilitate significant increases in industry-wide productivity.

Appendix A

Annotation Language and ArchCheckJ Typechecker[1]

This appendix describes the concrete annotation language, which uses existing language support for annotations, that I designed, and the typechecking tool that I implemented to typecheck the annotations.

A.1 Introduction

The previous implementation of ownership domains (Aldrich and Chambers 2004) used non-backwards compatible extensions of Java (ArchJava 2003). As a result, none of the rich tool support for Java programs was available to programs with ownership domain annotations[2].

In a previous case study (Abi-Antoun et al. 2007a), we discovered that adding ownership domain annotations to existing code often highlights refactoring opportunities. For instance, a lengthy domain parameter list is often an indication of tightly coupled code that could benefit from refactoring—such as extracting an interface and programming to that interface. It is unrealistic to assume that it is possible to refactor all such code prior to annotating it. In our experience, having access to refactoring tool support during the annotation process was invaluable. Using language extensions also makes it harder to partially and incrementally annotate existing code and thus conduct case studies on interesting systems. Finally, the previous tool used a modified research infrastructure (Bokowski and Spiegel 1998) that is no longer actively maintained and does not support Java generics as of this writing.

To address these adoptability challenges, we re-implemented the ownership domain type system using the annotation facility in Java 1.5 (Bloch 2004), so that Java programs with ownership annotations remain legal Java 1.5 programs. We also implemented the tool as a plugin to the Eclipse open source development environment that has become popular with researchers and practitioners (Goth 2005; Murphy et al. 2006).

[1] Portions of this chapter appeared in (Abi-Antoun and Aldrich 2007a).

[2] The Universes tools built on the Java Modeling Language (JML) infrastructure support both language extensions and stylized comments (Universes 2007).

We believe this improved tool support promotes the adoptability of the ownership domain technique by Java developers as follows. First, all the Eclipse tool support such as syntax highlighting, refactoring, etc., remains available to annotated programs. Second, using annotations makes it easier to support in a non-breaking way additional annotations such as external uniqueness (Clarke and Wrigstad 2003) or `readonly` (Dietl and Müller 2005). Third, using annotations provides the ability to incrementally and partially specify annotations on large code bases. Fourth, using annotations will make it possible to study the evolution of programs with ownership annotations, an area that has not received much attention—since no one will maintain a program with limited tool support. Finally, annotating existing code is difficult and time-consuming and tools are being developed to add annotations semi-automatically (Aldrich et al. 2002c; Cooper 2005). One of the benefits of using annotations over language extensions is that an inference algorithm cannot break an existing program by inserting potentially incorrect annotations.

We made the following design choices for the annotation system. First, we worked within the limits of Java 1.5 annotations (Bloch 2004), even though annotations may be more verbose than an elegantly designed language. Moreover, Java 1.5 annotations impose several restrictions, e.g., no annotations on generic type arguments. Other researchers have tried to eliminate some of these restrictions by proposing revisions of the language (Ernst and Coward 2006), but until such proposals are officially adopted, their prototype implementations are not Eclipse compatible, an important factor for adoptability. Second, to work around the Java 1.5 limitation of allowing annotations only on declarations, we consistently declare additional temporary variables and add annotations to them. This has worked well for new expressions, cast expressions (both implicit and explicit) and arguments for method and constructors. Third, checking ownership domain annotations generates only informational messages, i.e., no errors or warnings, and does not stop a developer from running the program. Fourth, we hard-code a minimal number of implicit defaults and provide a separate tool to supply explicit reasonable defaults to reduce the annotation burden. In the future, this tool can be replaced with a smarter annotation inference tool. Finally, the annotations are non-executable and do not impact the program's behavior[3]; unlike the earlier implementation, the current system does not include runtime checks. As a result, the annotation-based system is unsound at casts, but could be made sound using bytecode rewriting to add necessary dynamic checks.

This appendix is organized as follows: we describe the annotation language in Section A.2, the tool design in Section A.3 and other relevant features of the tool in Section A.4. We conclude with a discussion of the tool's limitations and some future work (Section A.5).

A.2 Annotation Design

In this section, we describe the concrete annotation syntax. For maximum flexibility and to work around some of the limitations of Java 1.5 annotations, all annotation values are strings. Annotations that are plural take values that are arrays of strings.

We illustrate the annotations using snippets from a canonical `Sequence` abstract data type, a common benchmark for ownership type systems. Within the `Sequence`, the `iters` ownership

[3] Annotations may increase the memory footprint and slow down class loading as a result, but no empirical data has been reported to date.

domain is used to hold `Iterator` objects that clients use to traverse the `Sequence`, and the default *private* `owned` ownership domain is used to hold the `Cons` cells in the linked list that is used to represent the `Sequence`. The full example is in Fig. A.1.

@Domains: declare public or private domains on a type.

- **Format**: $identifier$
- **Applies to**: type (class or interface).
- **Examples**: the following declares a private `owned` domain (`owned` is private by naming convention), and a public domain `iters` to store the `Iterator` objects of the `Sequence`.

```
@Domains({"owned","iters"})
class Sequence<T> {
...
}
```

@DomainParams: declare ordered domain parameters on a type or method domain parameters on a method.

- **Format**: $identifier$
- **Applies to**: type or method.
- **Examples**: `Sequence` declares a domain parameter `Towner` to hold its elements.

```
@DomainParams({"Towner"})
class Sequence<T> {
...
}
```

@DomainInherits: pass parameters to superclass or implemented interfaces.

- **Format**: $typename < parameter, \ldots >$
- **Applies to**: type (class or interface).
- **Examples**: the `Iterator` interface is also parameterized by the `Towner` domain parameter. Class `SeqIterator` inherits domain parameter `Towner` from interface `Iterator`, and adds the `list` parameter to access the `Cons` cells.

```
@DomainParams({"list", "Towner"})
@DomainInherits({"Iterator <Towner>"})
class SeqIterator<T> implements Iterator<T> {
...
}
```

@DomainLinks: declare domain links.

- **Format**: $fromDomainId$ -> $toDomainId$
- **Applies to**: type (class or interface).
- **Examples**: the `Sequence` gives `Iterator` objects in the `iters` domain permission to access objects in the `owned` domain, including the `Cons` cells.

```
@DomainLinks({..., "iters -> owned", ...})
class Sequence<T> {
...
}
```

@DomainAssumes: declare domain link assumptions.

- **Format**: $fromDomainId$ -> $toDomainIds$
- **Applies to**: type (class or interface).
- **Examples**: the `Sequence` assumes that the `owner` of the `Sequence` has access to the `Towner` domain containing the sequence elements.

```
@DomainAssumes("owner -> Towner") /* default */
class Sequence<T> {
...
}
```

@Domain: declare the domain, actual parameters and actual array parameters.

- **Format**: `annotation<domParams,...>[arrayParams,...]`
  - `annotation`: indicate a domain name (e.g., `owned`), one of the special alias types (e.g., `unique`), or a public domain of an object using a field access syntax (e.g., `seq.iters`);
  - `<domParams,...>`: specify actual domain parameters by order of formal domain parameters, at object creation and access sites;
  - `[arrayParams,...]`: in ownership domains, arrays have two ownership modifiers, one for the array object itself and one for the objects stored in the array. For variables of array type, this argument specifies the actual array parameters by order of array dimension (for multi-dimensional arrays).
  - **Applies to**: local variable declaration, field declaration, method formal parameter and method return value.
  - **Examples**: the following declares a `unique Iterator` object and binds the `list` domain parameter on `SeqIterator` to `owned` domain on `Sequence`, and the `Towner` domain parameter on `SeqIterator` to the parameter by the same name on `Sequence`.

    ```
    @Domain("unique<owned,Towner>")
    SeqIterator<T> it = new SeqIterator<T>(head);
    ```

  - **Examples**: a `lent` array of `shared Strings`:

    ```
    @Domain("lent[shared]")String args[];
    ```

@DomainReceiver: declare the domain of the receiver of a constructor or a method.

- **Format**: $identifier$
- **Applies to**: constructor or method.
- **Examples**:

```
@DomainReceiver("state")
void run() { ... }
```

A.3 Tool Design and Implementation

Two visitors on the Eclipse Abstract Syntax Tree (AST) typecheck the ownership domain annotations.

First pass. A first-pass visitor performs the following:

- **Identify problematic expressions:** a developer will need to replace each one with an equivalent construct, e.g., by declaring a local variable and adding the appropriate annotations to it;[4]

[4]Such an operation requires little effort when using the Eclipse refactoring (“Extract Local Variable”).

```
@Domains({"owned","iters"})
@DomainParams({"Towner"})
@DomainAssumes("owner -> Towner")
@DomainLinks({"owned->Towner", "iters->Towner", "iters->owned"})
class Sequence<T> {
  @Domain("owned<Towner>") Cons<T> head;
  void add(@Domain("Towner") T o) {
    @Domain("owned<Towner>")
    Cons<T> cons = new Cons<T>(o,head);
    head = cons;
  }
  @Domain("iters<Towner>") Iterator<T> getIter() {
    @Domain("iters<owned, Towner>") SeqIterator<T> it = new SeqIterator<T>(head);
    return it;
  }
}

@DomainParams({"Towner"})
@DomainAssumes("owner -> Towner")
class Cons<T> {
 @Domain("Towner") T obj;
 @Domain("owner<Towner>") Cons<T> next;

 Cons(@Domain("Towner") T obj, @Domain("owner<Towner>") Cons<T> next) {
   this.obj = obj;
   this.next = next;
 }
}

@DomainParams({"Towner"})
interface Iterator<T> {
  @Domain("Towner") T next();
  boolean hasNext();
}

@DomainParams({"list", "Towner"})
@DomainAssumes({"list -> Towner"})
@DomainInherits({"Iterator <Towner>"})
class SeqIterator<T> implements Iterator<T> {
  @Domain("list<Towner>") Cons<T> current;
  ...
  SeqIterator(@Domain("list<Towner>") Cons<T> head) {
    current = head;
  }
  public @Domain("Towner") T next() {
     @Domain("Towner") T obj2 = current.obj;
     current = current.next;
     return obj2;
   }
}

@Domains({"owned","state"})
class SequenceClient {
  final @Domain("owned<state>") Sequence<Integer> seq = new Sequence<Integer>();

  void run() {
    @Domain("state") Integer int5 = new Integer(5);
    seq.add(int5);
    @Domain("seq.iters<state>")Iterator<Integer> it = this.seq.getIter();
    while (it.hasNext()) {
      @Domain("state")Integer cur = it.next();
      ...
    }
  }
  ...
}
```

Figure A.1: A `Sequence` abstract data type with ownership domain annotations.

- **Read annotations from the AST:** the Java 1.5 annotations added to a program are part of the AST. The visitor locates the annotations nodes in the AST and parses their contents using a JavaCC (jav 2006) parser. The visitor also locates special block comments on method invocation expressions as described later. In addition, the visitor infers default annotations for some AST nodes that cannot be annotated, e.g., it implicitly defaults the `NullLiteral` AST node to `unique`. The visitor maps each AST node to an annotation structure in preparation for the second pass visitor which will typecheck the annotations;
- **Propagate local annotations:** the visitor propagates the explicit annotations from the AST nodes (for types, variables, and methods) to all the expression nodes in the AST, including translating formals to actuals.

Second pass. A second-pass visitor checks the annotations on each expression based on the static semantics of ownership domains. Checking the assignment rule requires a value flow analysis. A Live Variables Analysis (LVA) from a lightweight data flow analysis framework (Aldrich and Dickey 2006) that also uses the Eclipse AST, is invoked intra-procedurally at each method boundary using a separate visitor. The LVA analysis verifies that a `unique` pointer only has one non-`lent` read.

A.4 Additional Features

The tool offers the following additional features.

A.4.1 External Libraries

There are two approaches to support adding annotations to the standard Java libraries and other third-party libraries, one that involves annotating the library and pointing the tool to the annotated library and one that involves placing the annotations in external files. The earlier tool used the former approach (ArchJava 2003), but we adopted the latter approach this time since it does not require changing library or third-party code—which may not be available and when it is, tends to evolve separately. Other annotation-based systems adopted the same strategy (Qui 2006). The tool supports associating ownership domain annotations with any Java bytecode `.class` file using an external XML file, following the same annotation constructs described in Section A.2.

A.4.2 Generics

Our annotation system currently treats generic types as orthogonal to ownership domain parameters, so generic type parameters and arguments are added separately from ownership domain annotations—except that nested actual domains may need to be provided where applicable. Proponents of Generic Ownership (Potanin et al. 2006) argue that this leads to awkward syntax, which may be true. However, in our case studies annotating two 15,000-line Java programs including using generic types, we did not observe this to be a serious problem. Fig. A.2 illustrates the interaction between generics and ownership domains. The `Student` class is parameterized

```
@DomainParams({"state"})
class Student {
...
}
@DomainParams({"state"})
class Data ... {
  final @Domain("state<state<state>>")
  Sequence<Student> vStudent;

  @Domain("state<state>")Student
  getStudentRecord(@Domain("shared")String sSID) {
    @Domain("vStudent.iters<state<state>>")
    Iterator<Student> i = vStudent.getIter();
    while (i.hasNext()) {
      @Domain("state<state>")
      Student objStudent = i.next();
      ...
      }
    ...
    }
  }
```

Figure A.2: Adding annotations to generic code.

```
class Sequence<T> {
...
  @DomainParams("TTowner") /* Method domain parameter */
  @Domain("shared") /* Domain for return value */
  static <TT> String
    toString(@Domain("lent<TTowner>")Sequence<TT> seq) {
    ...
  }
  void dump() {
    @Domain("owned<shared>")
    Sequence<String> seq = ...;

    @Domain("shared")
    /* Provide <actuals...> using block comment */
    String str = Sequence.toString/*<state>*/(seq);
  }
}
```

Figure A.3: Declaring and binding method domain parameters.

by the `state` domain parameter. The `Data` class maintains a `Sequence` of `Student` objects and is also parameterized by `state`.

A.4.3 Method Domain Parameters

Java 1.5 annotations cannot be added at method invocation expressions. So we used block comments to specify the actual domains for a parameterized method (See Fig. A.3 for an example). Unfortunately, proposals to improve the Java 1.5 annotation facilities, e.g., (Ernst and Coward 2006), do not yet address adding annotations to such expressions.

```
while (objCourseFile.ready()) {
  this.vCourse.add(new Course(courseFile.readLine()));
}
/* ABOVE MUST BE REWRITTEN AS .... */
while (objCourseFile.ready()) {
 @Domain("shared")String line = courseFile.readLine();
 @Domain("state<state>")Course crs = new Course(line);
 this.vCourse.add(crs);
}
```

Figure A.4: Re-writing a new expression using a local variable.

A.4.4 Defaulting Tool

To reduce the annotation burden, we implemented a separate tool to add default annotations such as marking private fields as `owned`, method parameters as `lent`, and `Strings` as `shared`. However, an annotation added by the defaulting tool (e.g., `owned`) may need to be modified manually to supply actual domains for domain parameters (e.g., `owned<owned>`).

A.4.5 Special Annotations

Annotation 'owner'. We also added the special `owner` annotation, similar to `peer` in Universes (Dietl and Müller 2005). Using `owner` can often eliminate a domain parameter: e.g., in Fig. A.1, `Cons`'s `owner` is `Sequence`'s `owned`, `SeqIterator`'s `owner` is `Sequence`'s `iter`.

A.5 Tool Limitations and Future Work

Java 1.5 annotations suffer from the following limitations: (1) A declaration cannot have multiple annotations of the same annotation type; (2) Annotation types cannot have members of the their own type; (3) It is only legal to use single-member annotations for annotation types with multiple members, as long as one member is named `value`, and all other members have default values. Otherwise, the more verbose syntax is required, e.g., `@Name(first = "Joe", last = "Hacker")`; (4) Annotation types cannot extend any entity (class, interface or annotation); and (5) Annotations are allowed on type, field, variable and method declarations and not allowed on type parameters or method invocations.

The first restriction prevented us from using the `@Domain` annotation to specify both the annotation on the receiver and on the return type of a method. The second restriction prevented us from having shorthand constant annotations for the special alias types, e.g., `@owned` instead of `@Domain("owned")`: such constants cannot be used inside other annotations as in `@Domain(annotation = @owned, parameters = {@owned})`.

To avoid having multiple ways of indicating the same meaning, we use strings for all the annotations and require annotations of the form `@Domain("owned<owned>")`. Although developers may be more likely to introduce spelling mistakes in string annotations, the typechecker will catch these problems early enough. The third restriction, i.e., the lack of positional arguments, required the use of the verbose syntax `@Domains(publicDomains = {"d1", "d2"}, privateDomains = {"pda", "pdb"})`.

```
List vCourse = student.getRegisteredCourses(); for (int i=0; i<vCourse.size(); i++) {
 if (((Course) vCourse.get(i)).conflicts(course)) {
    ...
 }
} /* ABOVE MUST BE REWRITTEN AS ... */ @Domain("lent<state>") List vCourse = student.getRegisteredCourses(); for (int i=0;
i<vCourse.size(); i++) {
   @Domain("lent<state>")
   Course crs = (Course) vCourse.get(i);
   if (crs.conflicts(course)) {
       ...
   }
}
```

Figure A.5: Re-writing a cast expression using a local variable.

The final restriction and the current lack of annotation inference require converting some expressions to more verbose constructs by declaring local variables and annotating them. The most common such expressions were new expressions (Fig. A.4) and cast expressions (Fig. A.5).

We plan to address some of the following limitations:

- **Infer method domain parameters:** just as actual type arguments do not have to be passed to a generic method in Java, it may be possible to infer, in most cases, the actuals for method domain parameters based on the types of the actual arguments;
- **Allow suppressing warnings:** reflective code cannot be annotated successfully using ownership domains (Aldrich et al. 2002c). Because such code will always generate warnings, annotations to suppress spurious warnings can help reduce the number of persistent annotation warnings through which a developer has to wade.
- **Display annotations more concisely:** an Eclipse plug-in (Eisenberg and Kiczales 2007) can display verbose annotations using a simpler syntax for interactive editing while the analysis uses the same Java AST. We could use a similar approach to display the ownership domain Java 1.5 annotations using a simpler syntax similar to the one we used in this document.

A.6 Summary

We believe that re-implementing the ownership domain type system as backward-compatible Java 1.5 annotations, using the Eclipse infrastructure, significantly improved the tool support, and enabled us to conduct some of the largest case studies to date in applying ownership types to real object-oriented code.

For example, during our case studies, we often invoked the Eclipse refactoring tools to extract interfaces and infer generic types while adding the ownership domain annotations. This would not have been possible with the previous tool support.

The HillClimber subject system was annotated once using language extensions (Abi-Antoun et al. 2007a), and once using the annotation-based system. Comparing the number of hours across the two case studies would not be meaningful since the first case study added ownership annotations to the ArchJava version of HillClimber, HillClimberAJ, rather than the base Java version. Such a comparison also would not account for the learning effect of annotating roughly the same program twice. Still, anecdotally, we believe we were more productive

with the annotation-based system than with the earlier tool that used language extensions.

Appendix B

CryptoDB Architecture

Here, we reproduce the entire architectural model, in Acme (Garlan et al. 2000), for the CryptoDB case study. We provide both the family file, SyncFamily.acme (SectionB.1), which defines the architectural family that supports SCHOLIA, and the target architecture itself, CryptoDBTarget.acme (SectionB.2).

B.1 Architectural Style in Acme

This file defines the architectural family SyncFamily. The properties defined here are used by SCHOLIA for conformance analysis.

```
import  AS_GLOBAL_PATH/families/TieredFam.acme;

Family SyncFamily extends TieredFam with {

    analysis isSrcComponent(d1 : SyncCompT, conn : SyncConnT) : boolean =
        connected(conn, d1) and
        exists src : SyncUserT in conn.ROLES | exists put : SyncUseT in d1.PORTS |
            declaresType(src, SyncUserT) and declaresType(put, SyncUseT)
            and attached(src, put);

    analysis isDstComponent(d2 : SyncCompT, conn : SyncConnT) : boolean =
        connected(conn, d2) and
        exists dst : SyncProviderT in conn.ROLES | exists get : SyncProvideT in d2.PORTS |
            declaresType(dst, SyncProviderT) and declaresType(get, SyncProvideT)
            and attached(dst, get);

    analysis pointsTo(d1 : SyncCompT, d2 : SyncCompT) : boolean =
        exists conn : SyncConnT in self.CONNECTORS |
              isSrcComponent(d1, conn) and isDstComponent(d2, conn);

    Role Type SyncUserT extends userT with {
        Property syncStatus : int;
    }
    Component Type SyncCompT extends TierNodeT with {
        Property syncStatus : int;
```

```
        Property label : string;
        Property hasDetail : boolean;
        Property detailStatus : int;
        Property traceability : string;
    }
    Connector Type SyncConnT extends CallReturnConnT with {
        Property syncStatus : int;
        Property label : string;
        Property traceability : string;
        Property summary : int;
    }
    Port Type SyncUseT extends useT with {
        Property syncStatus : int;
    }
    Port Type SyncProvideT extends provideT with {
        Property syncStatus : int;
    }
    Role Type SyncProviderT extends providerT with {
        Property syncStatus : int;
    }
}
```

B.2 CryptoDB Target Architecture in Acme

This file defines the CryptoDB target architecture, including the constraints we discussed in Section 7.8.9 (Page 267).

```
import families/SyncFamily.acme;

System CryptoDBTarget : SyncFamily = new SyncFamily extended with {

    Component KeyVault : SyncCompT = new SyncCompT extended with {
        Port KeyVault : SyncProvideT = new SyncProvideT;
        Port KeyManager : SyncUseT = new SyncUseT;
        Port EngineWrapper : SyncUseT = new SyncUseT;

        Property label = ``KeyVault";
    }
    Component CryptoProvider : SyncCompT = new SyncCompT extended with {
        Port KeyManifest : SyncUseT = new SyncUseT;
        Port CryptoProvider : SyncProvideT = new SyncProvideT;
        Port CustomerManager : SyncUseT = new SyncUseT;
        Port EngineWrapper : SyncUseT = new SyncUseT;

        Property label = ``CryptoProvider";

        Representation CryptoProvider_Rep = {
            System CryptoProvider_Rep : SyncFamily = new SyncFamily extended with {
                Component ReceiptManager : SyncCompT = new SyncCompT extended with {
                    Port ReceiptManager : SyncProvideT = new SyncProvideT;
                    Port CryptoProvider : SyncUseT = new SyncUseT;
```

```
                Property label = ``ReceiptManager";
            }
            Component Encoder : SyncCompT = new SyncCompT extended with {
                Port CryptoProvider : SyncUseT = new SyncUseT;
                Port Encoder : SyncProvideT = new SyncProvideT;

                Property label = ``Encoder";
            }
        }
        Bindings {
            CustomerManager to ReceiptManager.CryptoProvider;
            EngineWrapper to Encoder.CryptoProvider;
        }
    }
}
Component KeyManager : SyncCompT = new SyncCompT extended with {
    Port KeyManifest : SyncUseT = new SyncUseT;
    Port KeyVault : SyncUseT = new SyncUseT;
    Port KeyManager : SyncProvideT = new SyncProvideT;

    Property label = ``KeyManager";
}
Component KeyManifest : SyncCompT = new SyncCompT extended with {
    Port KeyManifest : SyncProvideT = new SyncProvideT;
    Port KeyManager : SyncUseT = new SyncUseT;
    Port CryptoProvider : SyncUseT = new SyncUseT;

    Property label = ``KeyManifest";
}
Component EngineWrapper : SyncCompT = new SyncCompT extended with {
    Port EngineWrapper : SyncProvideT = new SyncProvideT;
    Port CryptoProvider : SyncUseT = new SyncUseT;
    Port KeyVault : SyncUseT = new SyncUseT;

    Property label = ``EngineWrapper";

    Representation EngineWrapper_Rep = {
        System EngineWrapper_Rep : SyncFamily = new SyncFamily extended with {
            Component Engine : SyncCompT = new SyncCompT extended with {
                Port Engine : SyncProvideT = new SyncProvideT;
                Port EngineWrapper : SyncUseT = new SyncUseT;

                Property label = ``Engine";
            }
        }
        Bindings {
            EngineWrapper to Engine.Engine;
            CryptoProvider to Engine.EngineWrapper;
        }
    }
}
```

```
Component CustomerManager : SyncCompT = new SyncCompT extended with {
    Port CustomerManager : SyncProvideT = new SyncProvideT;
    Port CryptoProvider : SyncUseT = new SyncUseT;
    Port CustomerInfo : SyncUseT = new SyncUseT;

    Property label = ``CustomerManager";

    Representation CustomerManager_Rep = {
        System CustomerManager_Rep : SyncFamily = new SyncFamily extended with {
            Component Receipts : SyncCompT = new SyncCompT extended with {
                Port Receipts : SyncProvideT = new SyncProvideT;
                Port CustomerManager : SyncUseT = new SyncUseT;

                Property label = ``Receipts";
            }
        }
        Bindings {
            CustomerManager to Receipts.Receipts;
            CryptoProvider to Receipts.CustomerManager;
        }
    }
}
Component CustomerInfo : SyncCompT = new SyncCompT extended with {
    Port CustomerManager : SyncUseT = new SyncUseT;
    Port CustomerInfo : SyncProvideT = new SyncProvideT;

    Property label = ``CustomerInfo";
}
Connector CustomerInfo_CustomerManager : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector CustomerManager_CustomerInfo : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector CustomerManager_CryptoProvider : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector CryptoProvider_CustomerManager : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector EngineWrapper_CryptoProvider : SyncConnT = new SyncConnT extended with {
    Role user : SyncUserT = new SyncUserT;
    Role provider : SyncProviderT = new SyncProviderT;
}
Connector CryptoProvider_EngineWrapper : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
```

```
Connector KeyVault_KeyManager : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyManager_KeyVault : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyManifest_KeyManager : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyManager_KeyManifest : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyManifest_CryptoProvider : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector CryptoProvider_KeyManifest : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector KeyVault_EngineWrapper : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Connector EngineWrapper_KeyVault : SyncConnT = new SyncConnT extended with {
    Role provider : SyncProviderT = new SyncProviderT;
    Role user : SyncUserT = new SyncUserT;
}
Attachment CryptoProvider.CustomerManager to CryptoProvider_CustomerManager.user;
Attachment CustomerManager.CustomerManager to CryptoProvider_CustomerManager.provider;
Attachment CustomerManager.CustomerManager to CustomerInfo_CustomerManager.provider;
Attachment CustomerInfo.CustomerInfo to CustomerManager_CustomerInfo.provider;
Attachment CustomerInfo.CustomerManager to CustomerInfo_CustomerManager.user;
Attachment KeyManifest.CryptoProvider to KeyManifest_CryptoProvider.user;
Attachment KeyVault.EngineWrapper to KeyVault_EngineWrapper.user;
Attachment EngineWrapper.EngineWrapper to CryptoProvider_EngineWrapper.provider;
Attachment EngineWrapper.CryptoProvider to EngineWrapper_CryptoProvider.user;
Attachment EngineWrapper.EngineWrapper to KeyVault_EngineWrapper.provider;
Attachment EngineWrapper.KeyVault to EngineWrapper_KeyVault.user;
Attachment CryptoProvider.EngineWrapper to CryptoProvider_EngineWrapper.user;
Attachment CryptoProvider.KeyManifest to CryptoProvider_KeyManifest.user;
Attachment KeyVault.KeyManager to KeyVault_KeyManager.user;
Attachment KeyManager.KeyVault to KeyManager_KeyVault.user;
Attachment KeyManifest.KeyManager to KeyManifest_KeyManager.user;
Attachment KeyManager.KeyManifest to KeyManager_KeyManifest.user;
Attachment CryptoProvider.CryptoProvider to CustomerManager_CryptoProvider.provider;
Attachment CryptoProvider.CryptoProvider to KeyManifest_CryptoProvider.provider;
Attachment CryptoProvider.CryptoProvider to EngineWrapper_CryptoProvider.provider;
```

```
    Attachment KeyManifest.KeyManifest to CryptoProvider_KeyManifest.provider;
    Attachment KeyManifest.KeyManifest to KeyManager_KeyManifest.provider;
    Attachment KeyManager.KeyManager to KeyManifest_KeyManager.provider;
    Attachment KeyManager.KeyManager to KeyVault_KeyManager.provider;
    Attachment KeyVault.KeyVault to EngineWrapper_KeyVault.provider;
    Attachment CustomerManager.CryptoProvider to CustomerManager_CryptoProvider.user;
    Attachment CustomerManager.CustomerInfo to CustomerManager_CustomerInfo.user;
    Attachment KeyVault.KeyVault to KeyManager_KeyVault.provider;
    Group KeyManagement = {
        Members {KeyManager}
    }
    Group CryptoConsumption = {
        Members {CustomerManager, CustomerInfo,
            CustomerManager_CustomerInfo, CustomerInfo_CustomerManager}
    }
    Group CryptoProvision = {
        Members {CryptoProvider, EngineWrapper,
            CryptoProvider_EngineWrapper, EngineWrapper_CryptoProvider}
    }
    Group KeyStorage = {
        Members {KeyManifest, KeyVault}
    }
    rule noVaultToManifest = invariant !pointsTo(KeyVault, KeyManifest);
    rule keyManagementAndEngineDisconnected = invariant
         forall c : Component in KeyManagement.MEMBERS | !connected(c, EngineWrapper);
    rule limitedVaultAccess = invariant forall c : SyncCompT in self.COMPONENTS |
        pointsTo(c, KeyVault) -> c.label == ``KeyManager" OR c.label == ``EngineWrapper";
}
```

Bibliography

JHotDraw. `www.jhotdraw.org`, 1996. Version 5.3.

JRM Tool. `http://jrmtool.sourceforge.net`, 2003.

Annotation File Utilities. `http://pag.csail.mit.edu/jsr308/annotation-file-utilities/`, 2006. Last accessed: Saturday, January 31, 2009.

JavaCC. `https://javacc.dev.java.net/`, 2006.

Marwan Abi-Antoun and Jonathan Aldrich. Ownership Domains in the Real World. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, pages 93–104, 2007a.

Marwan Abi-Antoun and Jonathan Aldrich. Compile-Time Views of Execution Structure Based on Ownership. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, pages 81–92, 2007b.

Marwan Abi-Antoun and Jonathan Aldrich. Checking and Measuring the Architectural Structural Conformance of Object-Oriented Systems. Technical Report CMU-ISRI-07-119R, Carnegie Mellon University, 2007c.

Marwan Abi-Antoun and Jonathan Aldrich. Static Conformance Checking of Runtime Architectural Structure. Technical Report CMU-ISR-08-132, Carnegie Mellon University, 2008a.

Marwan Abi-Antoun and Jonathan Aldrich. A Field Study in Static Extraction of Runtime Architectures. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 22–28, 2008b.

Marwan Abi-Antoun and Jonathan Aldrich. Static Extraction of Sound Hierarchical Runtime Object Graphs. In *Workshop on Types in Language Design and Implementation (TLDI)*, pages 51–64, 2009a.

Marwan Abi-Antoun and Jonathan Aldrich. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009b. To appear.

Marwan Abi-Antoun and Jonathan Aldrich. Practical Static Extraction and Conformance Checking of the Runtime Architecture of Object-Oriented Systems. Half-day tutorial at the SEI Architecture Technology User Network (SATURN). Available: `www.cs.cmu.edu/~mabianto/talks/09-SATURN_handout.pdf`, May 2009c.

Marwan Abi-Antoun and Jeffrey M. Barnes. Enforcing Conformance between Security Architecture and Implementation. Technical Report CMU-ISR-09-113, Carnegie Mellon University,

2009a.

Marwan Abi-Antoun and Jeffrey M. Barnes. Online addendum. http://www.cs.cmu.edu/ mabi-anto/cryptodb/, 2009b.

Marwan Abi-Antoun and Wesley Coelho. A Case Study in Incremental Architecture-Based Re-engineering of a Legacy Application. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 159–168, 2005.

Marwan Abi-Antoun, Jonathan Aldrich, Nagi Nahas, Bradley Schmerl, and David Garlan. Differencing and Merging of Architectural Views. In *Automated Software Engineering*, pages 47–58, 2006.

Marwan Abi-Antoun, Daniel Wang, and Peter Torr. Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security. Technical Report CMU-ISRI-06-124, Carnegie Mellon University, September 2006.

Marwan Abi-Antoun, Jonathan Aldrich, and Welsey Coelho. A Case Study in Re-engineering to Enforce Architectural Control Flow and Data Sharing. *J. Systems & Software*, 80(2):240–264, 2007a.

Marwan Abi-Antoun, Daniel Wang, and Peter Torr. Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security (Short Paper). In *Automated Software Engineering*, pages 393–396, 2007b.

Marwan Abi-Antoun, Jonathan Aldrich, Nagi Nahas, Bradley Schmerl, and David Garlan. Differencing and Merging of Architectural Views. *Automated Software Engineering*, 15(8):35–74, 2008.

Acme. Acme architectural description language. `www.cs.cmu.edu/~acme`, 2009.

AcmeStudio. AcmeStudio. `www.cs.cmu.edu/~acme/AcmeStudio/index.html`, 2009.

Rahul Agarwal and Scott D. Stoller. Type Inference for Parameterized Race-Free Java. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 149–160, 2004.

AgileJ. StructureViews. `www.agilej.com`, 2008.

Marcus Alanen and Ivan Porres. Difference and Union of Models. In *International Conference on the Unified Modeling Language, Modeling Languages and Applications*, pages 2–17, 2003.

Jonathan Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.

Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 1–25, 2004.

Jonathan Aldrich and David Dickey. The Crystal Data Flow Analysis Framework 2.0. `www.cs.cmu.edu/~aldrich/courses/654-sp06/`, 2006.

Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *International Conference on Software Engineering (ICSE)*, pages 187–197, 2002a.

Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning with ArchJava. In *European Conference on Object-Oriented Programming (ECOOP)*, 2002b.

Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330, 2002c.

Robert Allen and David Garlan. Formalizing Architectural Connection. In *International Conference on Software Engineering (ICSE)*, pages 71–80, 1994.

Manuel M. Ammann and Robert D. Cameron. Inter-Module Renaming and Reorganizing: Examples of Program Manipulation-in-the-Large. In *International Conference on Software Maintenance (ICSM)*, pages 354–361, 1994.

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A Framework for Implementing Pluggable Type Systems. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 57–74, 2006.

Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A Differencing Algorithm for Object-Oriented Programs. In *Automated Software Engineering*, pages 2–13, 2004.

ArchJava. ArchJava. `http://www.archjava.org/`, 2003.

Erik Arisholm, Lionel C. Briand, and Audun Foyen. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 324–341, 1996.

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2nd edition, 2003.

Kent Beck and Erich Gamma. JHotDraw – Patterns Applied (Tutorial). In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1997.

Colin J Bennett, Del Myers, Margaret-Anne Storey, Daniel M. German, David Ouellet, Martin Salois, and Philippe Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *J. Softw. Maint. Evol.*, 20(4):291–315, 2008.

Christophe Bidan and Valérie Issarny. Security Benefits from Software Architecture. In *Intl. Conf. on Coordination Languages and Models*, pages 64–80, 1997.

Martin Blech, Juan P. Carlino, J. Andrés Díaz-Pace, and Alvaro Soria. Keeping Design Documentation Updated through Synchronization of Use-Case-Maps with Implementation. In *Argentine Symposium on Software Engineering*, 2006.

Josh Bloch. *Effective Java*. Addison-Wesley, 2001.

Joshua Bloch. JSR 175: a Metadata Facility for the Java Programming Language. `http://jcp.org/en/jsr/detail?id=175`, 2004.

Boris Bokowski and André Spiegel. Barat – a Front-End for Java. Technical Report B-98-09, Freie Universität Berlin, 1998.

Boris Bokowski and Jan Vitek. Confined Types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a Case Study: its Extracted Software Architecture. In *International Conference on Software Engineering (ICSE)*, pages 555–563, 1999.

Chandrasekhar Boyapati. *SafeJava: a Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, February 2004.

Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *POPL*, pages 213–223, 2003a.

Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebee, and Martin Rinard. Ownership Types for Safe Region-Based Memory Mangement in Real-Time Java. In *Programming Language Design and Implementation (PLDI)*, 2003b.

Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL Component Model and its Support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: a System of Patterns*. John Wiley, 1996.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.

Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

Sylvain Chardigny, Abdelhak Seriai, Mourad Oussalah, and Dalila Tamzalit. Extraction of Component-Based Architecture from Object-Oriented Systems. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 285–288, 2008.

Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful Change Detection in Structured Data. In *ACM SIGMOD International Conference on Management of Data*, pages 26–37, 1997.

Ping H. Chen, Matt Critchlow, Akash Garg, Chris van der Westhuizen, and André van der Hoek. Differencing and Merging within an Evolving Product Line Architecture. In *Intl. Workshop on Software Product-Family Engineering*, pages 269–281, 2003.

Henrik Bærbak Christensen. Frameworks: Putting Design Patterns into Perspective. In *Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, 2004.

Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Equipping the Reflexion Method

with Automated Clustering. In *Working Conference on Reverse Engineering (WCRE)*, pages 89–98, 2005.

Dave Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 176–200, 2003.

David Clarke. *Object Ownership & Containment*. PhD thesis, University of New South Wales, July 2001.

David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64, 1998.

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.

Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty Years of Graph Matching in Pattern Recognition. *Int. J. Pattern Recognit. Artif. Intell.*, 18(3):265–298, 2004.

Will Cooper. Interactive Ownership Type Inference. Senior Thesis, Carnegie Mellon University, 2005.

Michelle L. Crane and Jürgen Dingel. Runtime Conformance Checking of Objects using Alloy. *Electronic Notes in Theoretical Computer Science*, 89(2):2–21, 2003.

Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A Highly-Extensible, XML-Based Architecture Description Language. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001.

Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. In *International Conference on Software Engineering (ICSE)*, pages 266–276, 2002.

Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. Adams re-trace: traceability link recovery via latent semantic indexing. In *International Conference on Software Engineering (ICSE)*, pages 839–842, 2008.

Andrea De De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4):13, 2007.

Wim De Pauw and Gary Sevitsky. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 116–134, 1999.

Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 326–337, 1993.

Wim De Pauw, Doug Kimelman, and John M. Vlissides. Modeling Object-Oriented Program Execution. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 163–182, 1994.

Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang.

Visualizing the Execution of Java Programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2002.

Yi Deng, Jiacun Wang, Jeffrey J. P. Tsai, and Konstantin Beznosov. An Approach for Modeling and Analysis of Security System Architectures. *IEEE Trans. on Knowledge and Data Engineering*, 15(5):1099–1119, 2003.

Elisabetta Di Nitto and David Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *International Conference on Software Engineering (ICSE)*, pages 13–22, 1999.

J. Andrés Díaz-Pace and Marcelo R. Campo. ArchMatE: from architectural styles to object-oriented models through exploratory tool support. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 117–132, 2005.

Peter J. Dickinson, Horst Bunke, Arek Dadej, and Miro Kraetzl. Matching Graphs with Unique Node Labels. *Pattern Analysis and Applications*, 7(3):243–254, December 2004.

Werner Dietl and Peter Müller. Universes: Lightweight Ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 28–53, 2007.

Edsger W. Dijkstra. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, 11(5):341–346, 1968.

Liliana Dobrica and Eila Niemel. A Survey on Software Architecture Analysis Methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002.

Stéphane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

W.J. Dzidek, E. Arisholm, and L.C. Briand. A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *IEEE Transactions on Software Engineering*, 34 (3):407–432, May-June 2008.

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

Alexander Egyed. Instant Consistency Checking for the UML. In *International Conference on Software Engineering (ICSE)*, pages 381–390, 2006.

Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and Continuous Checking of Structural Program Dependencies. In *International Conference on Software Engineering (ICSE)*, 2008.

T. Eisenbarth, R. Koschke, and G. Vogel. Static Trace Extraction. In *Working Conference on Reverse Engineering (WCRE)*, pages 128–137, 2002.

Andrew D. Eisenberg and Gregor Kiczales. Expressive Programs through Presentation Extension. In *Aspect-Oriented Software Development (AOSD)*, pages 73–84, 2007.

Hakan Erdogmus. Representing Architectural Evolution. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 159–177, 1998.

Michael D. Ernst and Danny Coward. JSR 308: Annotations on Java types. http://pag.csail.mit.edu/jsr308/, 2006.

Hoda Fahmy and Richard C. Holt. Software Architecture Transformations. In *International Conference on Software Maintenance (ICSM)*, page 88, 2000.

Loe Feijs, René L. Krikhaar, and Rob van Ommering. A Relational Approach to Support Software Architecture Analysis. *Software Pract. Experience*, 28(4), 1998.

Patrick Finnigan, Richard C. Holt, Ivan Kallas, Scott Kerr, Kostas Kontogiannis, Hausi A. Müller, John Mylopoulos, Stephen G. Perelgut, Martin Stanley, and Kerny Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

Roberto Fiutem and Giuliano Antoniol. Identifying Design-Code Inconsistencies in Object-Oriented Software: a Case Study. In *International Conference on Software Maintenance (ICSM)*, pages 94–102, 1998.

Cormac Flanagan and Stephen N. Freund. Dynamic Architecture Extraction. In *Workshop on Formal Approaches to Testing and Runtime Verification*, August 2006.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

Martin Fowler. UML Sketching Tools. http://martinfowler.com/bliki/UmlSketchingTools.html, 2004.

Robert Fuhrer, Frank Tip, Adam Kieżun, Julian Dolby, and Markus Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 71–96, 2005.

Erich Gamma. Advanced Design with Patterns and Java (Tutorial). In *European Conference on Java and Object Orientation (JAOO)*, 1998. JHotDraw v. 5.1.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software: Practice & Experience*, 30(11):1203–1233, 2000.

Juan Gargiulo and Spiros Mancoridis. Gadget: a Tool for Extracting the Dynamic Structure of Java Programs. In *Software Engineering and Knowledge Engineering*, 2001.

David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, I*, 1993.

David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural Description of Component-Based Systems. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Sci. Comput. Program.*, 44(1):23–49, 2002a.

David Garlan, Andrew Kompanek, and Shang-Wen Cheng. Reconciling the Needs of Architec-

tural Description with Object-Modeling Notations. *Science of Computer Programming*, 44: 23–49, 2002b.

Antoniol Giulio, Bruno Caprile, Alessandra Potrich, and Paolo Tonella. Design-code Traceability for Object-Oriented Systems. *Annals of Software Engineering*, 9(1-4):35–58, 2000.

Greg Goth. Beware the march of this IDE: Eclipse is overshadowing other tool techniques. *IEEE Software*, 22(4), 2005.

Aaron Greenhouse and John Boyland. An Object-Oriented Effects System. In *ecoop*, 1999.

Bernhard Gröne, Andreas Knöpfel, and Rudolf Kugel. Architecture Recovery of Apache 1.3 – a Case Study. In *International Conference on Software Engineering Research and Practice*, 2002.

Bernhard Gröne, Andreas Knöpfel, Rudolf Kugel, and Oliver Schmidt. The Apache Modeling Project. `http://www.fmc-modeling.org/projects/apache`, 2008.

Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating Objects with Confined Types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

John C. Grundy and John G. Hosking. Softarch: Tool Support for Integrated Software Architecture Development. *J. Softw. Eng. Klndg. Eng.*, 13(2), 2003.

Thomas Gschwind and Johann Oberleitner. Improving Dynamic Data Analysis with Aspect-Oriented Programming. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 259–268, 2003.

Yann-Gaël Guéhéneuc. A Reverse Engineering Tool for Precise Class Diagrams. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 28–41, 2004.

George Yanbing Guo, Joanne M. Atlee, and Rick Kazman. A Software Architecture Reconstruction Method. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 15–34, 1999.

Thomas Hächler. Applying the Universe Type System to an Industrial Application: Case Study. Master's thesis, Department of Computer Science, Federal Institute of Technology Zurich, 2005.

Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular Checking for Buffer Overflows in the Large. In *Intl. Conf. on Software Engineering*, pages 232–241, 2006.

Irit Hadar and Orit Hazzan. On the Contribution of UML Diagrams to Software System Comprehension. *Journal of Object Technology*, 3(1):143–156, 2004.

David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse Engineering to the Architectural Level. In *International Conference on Software Engineering (ICSE)*, pages 186–195, 1995.

Trent Hill, James Noble, and John Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *Journal of Visual Languages and Computing*, 13(3):319–339, 2002.

Adel Hlaoui and Shengrui Wang. A New Algorithm for Graph Matching with Application to Content-Based Image Retrieval. In *Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, pages 291–300, 2002.

Lorin Hochstein and Mikael Lindvall. Combating architectural degeneration: a survey. *Information & Software Technology*, 47(10):643–656, 2005.

Richard C. Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. GXL: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, 2006. `www.gupro.de/GXL/`.

H. James Hoover and Daqing Hou. Using SCL to Specify and Check Design Intent in Source Code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

Michael Howard and Steve Lipner. Inside the Windows Security Push. *IEEE Security and Privacy*, 1(1):57–61, 2003.

hyperCision Inc. jMetra. `www.hypercision.com`, 2008.

Anne Immonen and Eila Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008.

Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

Daniel Jackson and Martin Rinard. Software Analysis: a Roadmap. In *Conference on the Future of Software Engineering*, 2000.

Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering*, 27(2):156–169, 2001.

Catherine Blake Jaktman, John Leaney, and Ming Liu. Structural Analysis of the Software Architecture – a Maintenance Assessment Case Study. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 455–470, 1999.

Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing Interactions in Program Executions. In *International Conference on Software Engineering (ICSE)*, pages 360–370, 1997.

Tao Jiang, Lusheng Wang, and Kaizhong Zhang. Alignment of Trees – An Alternative to Tree Edit. In *Annual Symposium on Combinatorial Pattern Matching*, pages 75–86, 1994.

A. M. Jimenez. Change Propagation in the MDA: a Model Merging Approach. Master's thesis, University of Queesland, 2005.

J. Jürjens. *Secure Systems Development with UML*. Springer-Verlag, 2004.

Wolfram Kaiser. Become a programming Picasso with JHotDraw. JavaWorld, February 2001.

Rick Kazman and S. Jeromy Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Software Engineering*, 6(2):107–138, 1999.

Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture Reconstruction Guidelines, Third Edition. Technical Report CMU/SEI-2002-TR-034, Software Engineering Institute, 2002.

Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *International Conference on Software Engineering (ICSE)*, pages 226–235, 1999.

Kevin Kenan. *Cryptography in the Database*. Addison-Wesley, 2006. Accompanying code at

http://kevinkenan.blogs.com/downloads/cryptodb_code.zip.

Tahar Khammaci, Adel Smeda, and Mourad Oussalah. *Handbook of Software Engineering and Knowledge Engineering*, volume Vol 3: Recent Advances, chapter Coexistence of Object-Oriented Modeling and Architectural Description, pages 119–151. World Scientific Publishing, 2005.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

D. Kirk, M. Roper, and M. Wood. Identifying and Addressing Problems in Object-Oriented Framework Reuse. *Empirical Software Engineering*, 12(3):243–274, 2006.

Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, 1995.

Jens Knodel and Daniel Popescu. A Comparison of Static Architecture Compliance Checking Approaches. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2007.

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 (WCRE)*, pages 22–32, 2002.

Henk Koning, Claire Dormann, and Hans van Vliet. Practical Guidelines for the Readability of IT-Architecture Diagrams. In *International Conference on Computer Documentation (SIGDOC)*, pages 90–99, 2002.

K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for design concept localization. In *Working Conference on Reverse Engineering (WCRE)*, pages 96–103, 1995.

Rainer Koschke. Architecture Reconstruction: Tutorial on Reverse Engineering to the Architectural Level. In Andrea De Lucia and Filomena Ferrucci, editors, *International Summer School on Software Engineering*, pages 140–173, 2008.

Rainer Koschke and Daniel Simon. Hierarchical Reflexion Models. In *Working Conference on Reverse Engineering (WCRE)*, page 36, 2003.

Kai Koskimies and Hanspeter Mössenböck. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In *International Conference on Software Engineering (ICSE)*, pages 366–375, 1996.

Christian Kramer and Lutz Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. *Working Conference on Reverse Engineering (WCRE)*, page 208, 1996.

René L. Krikhaar. Reverse Architecting Approach for Complex Systems. In *International Conference on Software Maintenance (ICSM)*, pages 4–11, 1997.

René L. Krikhaar, A. Postma, A. Sellink, M. Stroucken, and C. Verhoef. A Two-Phase Process for Software Architecture Improvement. In *International Conference on Software Maintenance (ICSM)*, pages 371–380, 1999.

Neel Krishnaswami and Jonathan Aldrich. Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages. In *Programming Language Design and Implementation (PLDI)*, pages 96–106, 2005.

Philippe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

Bruno Laguë, Charles Leduc, André Le Bon, Ettore Merlo, and Michel Dagenais. An Analysis Framework for Understanding Layered Software Architectures. *International Workshop on Program Comprehension (IWPC)*, 1998.

Patrick Lam and Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 275–302, 2003.

Danny B. Lange and Yuichi Nakamura. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 342–357, 1995.

Lattix Inc. LDM tool. `http://www.lattix.com/`, 2008.

Seonah Lee, Gail C. Murphy, Thomas Fritz, and Meghan Allen. How Can Diagramming Tools Help Support Programming Activities? In *VL/HCC*, pages 246–249, 2008.

Karl J. Lieberherr and Ian M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, 6(5), 1989.

Mikael Lindvall and Kristian Sandahl. Practical Implications of Traceability. *Softw. Pract. Exper.*, 26(10):1161–1180, 1996.

Yin Liu and Ana Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *International Conference on Software Engineering (ICSE)*, pages 323–332, 2007.

Yu Liu and Scott Smith. Pedigree Types. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2008.

Torsten Lodderstedt, David A. Basin, and Jürgen Doser. SecureUML: a UML-Based Modeling Language for Model-Driven Security. In *Intl. Conference on the Unified Modeling Language*, pages 426–441, 2002.

Yi Lu and John Potter. Protecting Representation with Effect Encapsulation. In *POPL*, pages 359–371, 2006.

David C. Luckham and James Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

Kin-Keung Ma and Jeffrey S. Foster. Inferring Aliasing and Encapsulation Properties for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

Neel Madhav. Testing Ada 95 Programs for Conformance to Rapide Architectures. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 123–134, 1996.

Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. In *Foundations of Software Engineering (FSE)*, pages 3–14, 1996.

Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying Distributed Software

Architectures. In *European Software Engineering Conference*, pages 137–153, 1995.

Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.

Andrew J. Malton and Richard C. Holt. Boxology of NBA and TA: a basis for understanding software architecture. In *Working Conference on Reverse Engineering (WCRE)*, pages 187–195, 2005.

S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *International Conference on Software Maintenance (ICSM)*, pages 50–59, 1999.

David Mandelin, Doug Kimelman, and Daniel Yellin. A Bayesian Approach to Diagram Matching with Application to Architectural Models. In *International Conference on Software Engineering (ICSE)*, 2006.

Onaiza Maqbool and Haroon Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

Joseph F. Maranzano, Sandra A. Rozsypal, Gus H. Zimmerman, Guy W. Warnken, Patricia E. Wirth, and David M. Weiss. Architecture Reviews: Practice and Experience. *IEEE Softw.*, 22 (2):34–43, 2005.

Nenad Medvidovic and Vladimir Jakobac. Using Software Evolution to Focus Architectural Recovery. *Automated Software Engineering*, 13(2):225–256, 2006.

Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), 2000.

Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Foundations of Software Engineering (FSE)*, 1996.

Akhil Mehra, John Grundy, and John Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Automated Software Engineering*, 2005.

Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity Flooding: a Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *International Conference on Data Engineering*, pages 117–128, 2002.

Nabor C. Mendonça and Jeff Kramer. An Approach for Recovering Distributed System Architectures. *Automated Software Engineering*, 8(3-4):311–354, 2001.

Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. In *Proc. Int'l Workshop on Graph and Model Transformation*, 2005.

B.T. Messmer. *Efficient Graph Matching Algorithms for Preprocessed Model Graphs*. PhD thesis, University of Bern, 1996.

Ana Milanova. Static Inference of Universe Types. In *Intl. Workshop on Aliasing, Confinement*

and Ownership in Object-Oriented Programming (IWACO), 2008.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Constructing Precise Object Relation Diagrams. In *International Conference on Software Maintenance (ICSM)*, pages 586–595, 2002.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-To Analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.

Nick Mitchell. The Runtime Structure of Object Ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 57–64, 2006.

Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Making Sense of Large Heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, 2009.

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

Mark Moriconi, Xiaolei Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.

Mark Moriconi, Xiaolei Qian, R. A. Riemenschneider, and Li Gong. Secure Software Architectures. In *IEEE Symposium on Security and Privacy*, page 84, 1997.

Henry Muccini, Marcio S. Dias, and Debra J. Richardson. Towards Software Architecture-Based Regression Testing. In *Workshop on Architecting Dependable Systems*, pages 1–7, 2005.

Hausi Müller and Karl Klashinsky. Rigi – a System for Programming-In-The-Large. In *International Conference on Software Engineering (ICSE)*, pages 80–86, 1988.

Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A Reverse-Engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

Peter Müller and Arnd Poetzsch-Heffter. Universes: a Type System for Controlling Representation Exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*, 1999.

Peter Müller and Arsenii Rudich. Ownership Transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

Gail C. Murphy. *Lightweight Structural Summarization as an Aid to Software Evolution*. PhD thesis, University of Washington, 1996.

Gail C. Murphy and David Notkin. Lightweight Source Model Extraction. In *Foundations of Software Engineering (FSE)*, pages 116–127, 1995.

Gail C. Murphy and David Notkin. Reengineering with reflexion models: a case study. *Computer*, 30(8):29–36, 1997.

Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Transactions on Software Engineering*, 27

(4):364–380, 2001.

Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 23(4), 2006.

Stefan Nägeli. Ownership in Design Patterns. Master's thesis, Department of Computer Science, Federal Institute of Technology Zurich, 2006.

Nagi H. Nahas. Algorithms for the Comparison of Unordered Labeled Trees. Master's thesis, American University of Beirut, Beirut, Lebanon, May 2009.

J. Nielsen and R.L. Mack, editors. *Usability Inspection Methods*. John Wiley & Sons, 1994.

Eugen C. Nistor, Justin R. Erenkrantz, Scott A. Hendrickson, and André van der Hoek. ArchEvol: Versioning Architectural-Implementation Relationships. In *International Workshop on Software Configuration Management*, pages 99–111, 2005.

James Noble. Visualising Objects: Abstraction, Encapsulation, Aliasing, and Ownership. In *Revised Lectures on Software Visualization, International Seminar*, pages 58–72, 2002.

James Noble, Jan Vitek, and John Potter. Flexible Alias Protection. In *European Conference on Object-Oriented Programming (ECOOP)*, 1998.

Object Technology International, Inc. Eclipse Platform Technical Overview. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`, 2003.

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

Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams using the Java Debug Interface (JDI). In *Revised Lectures on Software Visualization, International Seminar*, pages 176–190, 2002.

OGJ. Ownership Generic Java (OGJ). `www.mcs.vuw.ac.nz/~alex/ogj/`, 2005.

Dirk Ohst, Michael Welle, and Udo Kelter. Differences between Versions of UML Diagrams. In *European Software Engineering Conference (ESEC)/Foundations of Software Engineering (FSE)*, pages 227–236, 2003.

Rocco Oliveto, Giuliano Antoniol, Andrian Marcus, and Jane Hayes. Software artefact traceability: the never-ending challenge. In *International Conference on Software Maintenance (ICSM)*, pages 485–488, 2007.

OMG. Unified Modeling Language (UML), 2008.

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

Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-Based Runtime Software Evolution. In *International Conference on Software Engineering (ICSE)*, 1998.

Michael J. Pacione, Marc Roper, and Murray Wood. A Novel Software Visualisation Model to Support Software Comprehension. In *Working Conference on Reverse Engineering (WCRE)*, pages 70–79, 2004.

PBS. PBS: The Portable Bookshelf. `http://www.swag.uwaterloo.ca/pbs/`, 2000.

Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture.

SIGSOFT Softw. Eng. Notes, 17(4):40–52, 1992.

David Pichardie. Constraint based analysis for Java. `www.irisa.fr/lande/teaching/PAS/pointsto.pdf`, 2008.

David Poole and Alan Macworth. CISpace: Tools for learning Computational Intelligence. `http://www.cs.ubc.ca/labs/lci/CIspace/`, 2001.

Andre Postma. A Method for Module Architecture Verification and its Application on a Large Component-Based System. *Information and Software Technology*, 45(4):171–194, 2003.

Alex Potanin. *Generic Ownership: A Practical Approach to Ownership and Confinement in Object-Oriented Programming Languages*. PhD thesis, Victoria University of Wellington, 2007.

Alex Potanin, James Noble, and Robert Biddle. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, April 2004.

Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic Ownership for Generic Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–412, 2006.

John Potter, James Noble, and David Clarke. The Ins and Outs of Objects. In *Australian Software Engineering Conference*, pages 80–89, 1998.

Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: a Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases. In *International Conference on Software Maintenance (ICSM)*, pages 188–197, 2004.

Derek Rayside and Lucy Mendel. Object Ownership Profiling: a Technique for Finding and Fixing Memory Leaks. In *Automated Software Engineering*, 2007.

Derek Rayside, Lucy Mendel, Robert Seater, and Daniel Jackson. An Analysis and Visualization for Revealing Object Sharing. In *Eclipse Technology eXchange (ETX)*, pages 11–15, 2005.

Derek Rayside, Lucy Mendel, and Daniel Jackson. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Workshop on Dynamic Analysis (WODA)*, pages 57–64, 2006.

Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – a Tool Suite for Program Analysis and Reverse Engineering. In *International Conference on Reliable Software Technologies (Ada-Europe)*, pages 71–82, 2006.

Trygve Reenskaug. Thing-Model-View-Editor – an Example from a planning system. Technical note, Xerox PARC. Available at: `http://heim.ifi.uio.no/~trygver/mvc/index.html`, 1979.

Trygve Reenskaug. *Working with objects: the OOram Software Engineering Method*. Manning/Prentice Hall, 1996.

Trygve Reenskaug. The Common Sense of Object Orientated Programming. `http://heim.ifi.uio.no/~trygver/2008/commonsense.pdf`, 2008.

Steven P. Reiss and Manos Renieris. Jove: Java as it Happens. In *ACM Symposium on Software Visualization*, pages 115–124, 2005.

Jie Ren and Richard Taylor. A Secure Software Architecture Description Language. In *Workshop*

on Softw. Security Assurance Tools, Techniques, and Metrics, 2005.

Tamar Richner and Stephane Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *International Conference on Software Maintenance (ICSM)*, pages 13–22, 1999.

Dirk Riehle. *Framework Design: a Role Modeling Approach*. PhD thesis, Federal Institute of Technology Zurich, 2000.

Roshanak Roshandel, André van der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. Mae – a System Model and Environment for Managing Architectural Evolution. *ACM Transactions on Software Engineering and Methodology*, 13(2):240–276, 2004.

Roshanak Roshandel, Nenad Medvidovic, and Leana Golubchik. A Bayesian Model for Predicting Reliability of Software Systems at the Architectural Level. In *International Conference on Quality of Software Architectures*, 2007.

Jacek Rosik, Andrew Le Gear, Jim Buckley, and Muhammad Ali Babar. An Industrial Case Study of Architecture Conformance. In *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 80–89, 2008.

James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

John Rushby, Sam Owre, and Natarajan Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9), 1998.

David Saff and Michael D. Ernst. Continuous Testing in Eclipse. In *International Conference on Software Engineering (ICSE)*, pages 668–669, 2005.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *POPL*, pages 105–118, 1999.

Maher Salah and Spiros Mancoridis. A Hierarchy of Dynamic Software Views: From Object-Interactions to Feature-Interactions. In *International Conference on Software Maintenance (ICSM)*, 2004.

Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using Dependency Models to Manage Complex Software Architecture. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

Santonu Sarkar, Girish Maskeri Rama, and Shubha R. A Method for Detecting and Measuring Architectural Layering Violations in Source Code. In *Asia Pacific Software Engineering Conference*, pages 165–172, 2006.

Kamran Sartipi and Kostas Kontogiannis. A User-Assisted Approach to Component Clustering. *Journal of Software Maintenance*, 15(4):265–295, 2003a.

Kamran Sartipi and Kostas Kontogiannis. On Modeling Software Architecture Recovery as Graph Matching. In *International Conference on Software Maintenance (ICSM)*, pages 224–234, 2003b.

Kamran Sartipi and Kostas Kontogiannis. *Managing Corporate Information Systems Evolution and Maintenance*, chapter Software Architecture Analysis and Reconstruction. Idea Group

Publishing, 2004.

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

Jan Schäfer, Markus Reitz, Jean-Marie Gaillourdet, and Arnd Poetzsch-Heffter. Linking Programs to Architectures: An Object-Oriented Hierarchical Software Model based on Boxes. In *The Common Component Modeling Example: Comparing Software Component Models*, LNCS, pages 238–266. Springer, 2008.

Reinhard Schauer and Rudolf K. Keller. Pattern Visualization for Software Comprehension. In *International Workshop on Program Comprehension (IWPC)*, page 4, 1998.

Bradley Schmerl and David Garlan. AcmeStudio: Supporting Style-Centered Architecture Development. In *International Conference on Software Engineering (ICSE)*, pages 704–705, 2004.

Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering Architectures from Running Systems. *IEEE Transactions on Software Engineering*, 32(7): 454–466, 2006.

Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Architecture-Oriented Visualization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 389–405, 1996a.

Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring Compliance of a Software System with its High-Level Design Models. In *International Conference on Software Engineering (ICSE)*, pages 387–396, 1996b.

D. Shasha and K Zhang. Approximate Tree Pattern Matching. In A. Apostolico and Eds Galil, Z., editors, *Pattern Matching Algorithms*. Oxford University Press, 1997.

Mary Shaw and Paul Clements. The Golden Age of Software Architecture. *IEEE Softw.*, 23(2): 31–39, 2006.

Mary Shaw and David Garlan. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.

Mati Shomrat and Amiram Yehudai. Obvious or not? Regulating Architectural Decisions using Aspect-Oriented Programming. In *Aspect-Oriented Software Development (AOSD)*, pages 3–9, 2002.

Forrest Shull, Filippo Lanubile, and Victor R. Basili. Investigating Reading Techniques for Object-Oriented Framework Learning. *IEEE Transactions on Software Engineering*, 26(11): 1101–1118, 2000.

Vineet Sinha, David R. Karger, and Rob Miller. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. In *VL/HCC*, pages 187–194, 2006.

Michael P. Smith and Malcolm Munro. Runtime Visualisation of Object Oriented Software. In

VISSOFT, 2002.

Dilip Soni, Robert L. Nord, and Christine Hofmeister. Software Architecture in Industrial Applications. In *International Conference on Software Engineering (ICSE)*, pages 196–207, 1995.

Tim Souder, Spiros Mancoridis, and Maher Salah. Form: a Framework for Creating Views of Program Executions. In *International Conference on Software Maintenance (ICSM)*, 2001.

George Spanoudakis and Andrea Zisman. *Handbook of Software Engineering and Knowledge Engineering*, volume Vol 3: Recent Advances, chapter Software Traceability: A Roadmap, pages 395–428. World Scientific Publishing, 2005.

André Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.

Diomidis Spinellis. On the Declarative Specification of Models. *IEEE Software*, 20(2):94–96, March/April 2003.

Bridget Spitznagel and David Garlan. Architecture-Based Performance Analysis. In *Conference on Software Engineering and Knowledge Engineering*, 1998.

Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 59–76, 2005.

Margaret-Anne Storey, Casey Best, and Jeff Michaud. SHriMP Views: An Interactive Environment for Exploring Java Programs. In *International Workshop on Program Comprehension (IWPC)*, pages 111–112, 2001.

Margaret-Anne D. Storey, Hausi A. Müller, and Kenny Wong. Manipulating and Documenting Software Structures. In P. Eades and K. Zhang, editors, *Software Visualization*, 1998.

Margaret-Anne D. Storey, Frank D. Fracchia, and Hausi A. Müller. Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *J. Systems & Software*, 44(3), 1999.

Sun Microsystems. J2EE Tutorials. Dukes Bank. `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Eba` 2006.

Tarja Systä, Ping Yu, and Hausi Müller. Analyzing Java software by combining metrics and program visualization. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 199–208, 2000.

Peter Tabeling and Bernhard Gröne. Mappings between Object-Oriented Technology and Architecture-Based Models. In *Software Engineering Research and Practice*, pages 568–574, 2003.

Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Jr. Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22 (6):390–406, 1996.

Alexandru Telea, Alessandro Maccari, and Claudio Riva. An Open Visualization Toolkit for Reverse Architecting. In *International Workshop on Program Comprehension (IWPC)*, pages 3–10, 2002.

Paolo Tonella and Alessandra Potrich. Static and Dynamic C++ Code Analysis for the Recovery of the Object Diagram. In *International Conference on Software Maintenance (ICSM)*, pages 54–63, 2002.

Paolo Tonella and Alessandra Potrich. Reverse Engineering of the Interaction Diagrams from C++ Code. In *International Conference on Software Maintenance (ICSM)*, pages 159–168, 2003.

Paolo Tonella and Alessandra Potrich. *Reverse Engineering of Object Oriented Code (Monographs in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

Paolo Tonella, Giuliano Antoniol, Roberto Fiutem, and Ettore Merlo. Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In *International Conference on Software Engineering (ICSE)*, pages 433–443, 1997.

Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. Empirical Studies in Reverse Engineering: State of the Art and Future Trends. *Empirical Software Engineering*, 12(5): 551–571, 2007.

Peter Torr. Demystifying the Threat-Modeling Process. *IEEE Security and Privacy*, 3(5):66–70, 2005.

Andrea Torsello, Dzena Hidovic-Rowe, and Marcello Pelillo. Polynomial-Time Metrics for Attributed Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(7):1087–1099, 2005.

Christopher J. Turner, T.C. Nicholas Graham, Christopher Wolfe, Julian Ball, David Holman, Hugh D. Stewart, and Arthur G. Ryman. Visual Constraint Diagrams: Runtime Conformance Checking of UML Object Models versus Implementations. In *Automated Software Engineering*, pages 271–276, 2003.

Roseanne Tesoriero Tvedt, Patricia Costa, and Mikael Lindvall. Does the Code Match the Design? A Process for Architecture Evaluation. In *International Conference on Software Maintenance (ICSM)*, pages 393–401, 2002.

Vassilios Tzerpos and Richard C. Holt. A Hybrid Process for Recovering Software Architecture. In *Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, page 38, 1996.

Universes. Universes Tools. `www.sct.ethz.ch/research/universes/tools/`, 2007.

Christopher van der Westhuizen and André van der Hoek. Understanding and Propagating Architectural Changes. In *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 95–109, 2002.

Hylke W. van Dijk, Bas Graaf, and Rob Boerman. On the Systematic Conformance Check of Software Artefacts. In *European Workshop on Software Architecture (EWSA)*, pages 204–221, 2005.

Allison Waingold. Automatic Extraction of Abstract Object Models. Master's thesis, Department

of Electrical Engineering and Computer Science, MIT, 2001.

Allison Waingold and Robert Lee. SuperWomble Manual. `http://sdg.lcs.mit.edu/womble/`, 2002.

Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing Dynamic Software System Information through High-Level Models. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 271–283, 1998.

Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. *International Conference on Data Engineering*, pages 519–530, 2003.

Dietl Werner and Peter Müller. Exceptions in Ownership Type Systems. In *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004.

Dietl Werner and Peter Müller. Runtime Universe Type Inference. In *Intl. Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, 2007.

David S. Wile. Revealing Component Properties through Architectural Styles. *J. Systems & Software*, 65(3), 2003.

Lloyd G. Williams and Connie U. Smith. Performance evaluation of software architectures. In *International Workshop on Software and Performance (WOSP)*, pages 164–177, 1998.

Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural Redocumentation: a Case Study. *IEEE Software*, 12(1):46–54, 1995.

Alisdair Wren. Ownership Type Inference. Master's thesis, Department of Computing, Imperial College, 2003.

Zhenchang Xing and Eleni Stroulia. UMLDiff: an Algorithm for Object-Oriented Design Differencing. In *Automated Software Engineering*, pages 54–65, 2005.

Guoqing Xu and Atanas Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 225–236, 2008.

Kaizhong Zhang and Tao Jiang. Some MAX SNP-Hard Results Concerning Unordered Labeled Trees. *Information Processing Letters*, 49(5):249–254, 1994.