Static Extraction of Object-Oriented Runtime Architectures[1]

Marwan Abi-Antoun **Jonathan Aldrich**

March 2008
CMU-ISR-08-127

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

For many object-oriented systems, it is useful to have a runtime architecture, that shows networks of objects. But it is hard to statically extract runtime object graphs that provide architectural abstraction from an arbitrary program written in a general purpose language that follows common design idioms.
Previous approaches extract low-level non-hierarchical object graphs that do not provide architectural abstraction, change the language too radically for many existing implementations, or use a dynamic analysis. Static analysis, which takes all possible executions into account, is essential to extract a sound architecture, one that reveals all entities and relations that could possibly exist at runtime.
Ownership domain type annotations specify in code architectural intent related to object encapsulation and communication. We propose a static analysis that leverages such types and extracts a hierarchical approximation of all possible runtime object graphs. The representation provides architectural abstraction, first by ownership hierarchy, and then by types. We proved core soundness results for the technique and evaluated it on several extended examples of medium-sized representative programs that we annotated manually.

[1]This report updates the following paper: Abi-Antoun, M. and Aldrich, J. Compile-Time Views of Execution Structure Based on Ownership. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007. A shorter and updated version of this document will appear as: Abi-Antoun, M. and Aldrich, J. Static Extraction of Sound Hierarchical Runtime Object Graphs. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, January 2009.

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

Keywords: runtime architecture, architecture extraction, architecture recovery, component-and-connector views, ownership domains, object graphs

Contents

List of Figures

"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 [18]).

1 Introduction

Many architectural views are needed to describe a software system. The *code architecture* or *module view* organizes code entities in terms of classes, packages, layers and modules [13], and is useful for studying properties such as maintainability. Another useful view is the *runtime architecture* of a system [13]. A runtime architecture, also known as a Component-and-Connector (C&C) view, models runtime entities and their potential interactions[1].

A runtime *component* is a unit of computation and state that has a runtime presence. Thus, in an object-oriented system, a component is an object or a group of objects [13]. A *connector* is an abstraction of a runtime interaction. In an object-oriented system, a connector models one or more object relations. Architectures often organize components into *tiers*. A *tier* is a conceptual partitioning of functionality[2]. Finally, architectures are often hierarchical whereby a component can have a nested sub-architecture consisting of lower-level components and connectors [30].

While the above definitions are consistent with formal Architecture Description Languages (ADLs) and software architecture research, developers intuitively draw such runtime architectures on whiteboards. Figure 1 is a runtime architecture drawn by the developer of a system we study in Section 6.3.

Architectural-level analyses for properties such as performance, reliability or security require runtime views. Moreover, having an up-to-date as-built runtime architecture enables checking the conformance of a system with its as-designed architecture. Despite receiving much research attention, architectural extraction remains a hard problem.

Recovering meaningful runtime architectures statically is hard for object-oriented systems since their runtime architecture often bears little resemblance to their code architecture. In fact, most recovery approaches employ a mix of static and dynamic information. To simplify the problem of relating architecture to code, previous research mandated specific implementation frameworks [30] or extended the language to specify a component-and-connector architecture directly in code [6, 43]. Such proposals require re-engineering existing implementations. Our goal in this paper is to support existing object-oriented languages, design idioms and existing libraries and frameworks.

Intuitively, many have preferred dynamic analyses to extract the as-built runtime architecture. Such an analysis monitors one or more program runs and shows snapshots of the system's runtime architecture for those runs [45, 15, 44]. But these descriptions are partial and cover only a few representative interactions between objects, based on particular inputs and exercised use cases. A true architecture is meant to capture a complete description of the system's runtime structure. To meet this goal, a static analysis is preferred.

A static analysis must also be *sound* and not fail to reveal entities and relationships that actually exist at runtime. For instance, an architectural-level security analysis requires a complete architectural description to handle the worst, not the typical, case of runtime component communication.

In this paper, we propose the static extraction of a runtime view of an object-oriented system with a two-pronged approach: (a) assume that annotations encode and enforce the architectural intent in code, as discussed by Aldrich et al. [5, 3]; and (b) leverage the annotations in a static analysis to extract a sound[3]runtime architecture from an annotated program. Our contributions in this paper are:

- A static analysis for extracting an instance-based hierarchical runtime architecture based on program annotations;
- A soundness proof of the extracted architecture;
- An evaluation of the analysis on several representative medium-sized object-oriented systems.

Outline. The paper is organized as follows. We first discuss the requirements on a runtime architecture (Section 2) and how the annotations help with architectural extraction. In Section 3, we describe the core

[1] A related notion is that of an *object diagram*, a diagram of object structures which shows object instances exclusively. Gamma et al. use object diagrams extensively to explain the Gang of Four design patterns [18].

[2] A *layer* denotes a partition in the *code architecture* or a *module view* [13]; it can be represented as a package and enforced using dependency rules [42]. A *tier* denotes a partition in a *runtime view* [13].

[3] In the software architecture literature, a *sound* architecture often means an architecture with desirable quality attributes. In this paper, a sound architecture shows the as-is actual architecture with tight coupling between components or other undesirable quality attributes notwithstanding. The analysis does not give any judgment about the architecture of a system.

analysis informally. In Section 4, we describe the analysis formally and prove key soundness theorems. In Section 5, we discuss abstraction by types. Section 6 presents highlights of our evaluation on several real systems. We conclude with a discussion (Section 7) and a survey of related work (Section 8).

2 Overall Strategy

A *runtime object graph* represents a running object-oriented program where nodes correspond to runtime objects, and edges correspond to relations between objects. A sound runtime architecture must statically approximate all the runtime object graphs that any program run may generate.

Existing static analyses that extract a system's execution structure produce low-level, non-hierarchical object graphs that explain runtime interactions in detail but convey little architectural abstraction [33, 21, 46]. Figure 2 shows the output of a static object graph analysis, WOMBLE [21], on a 15,000-line program, JHotDraw (`http://www.jhotdraw.org` (Version 5.3)). Low-level objects such `Dimension` and `Rectangle` appear at the same level as the root application object, `JavaDrawApp`. Such a view is a far cry from what a developer might draw for a runtime architecture (See Figure 1). The PANGAEA output for JHotDraw is even more complex (Fig. 3).

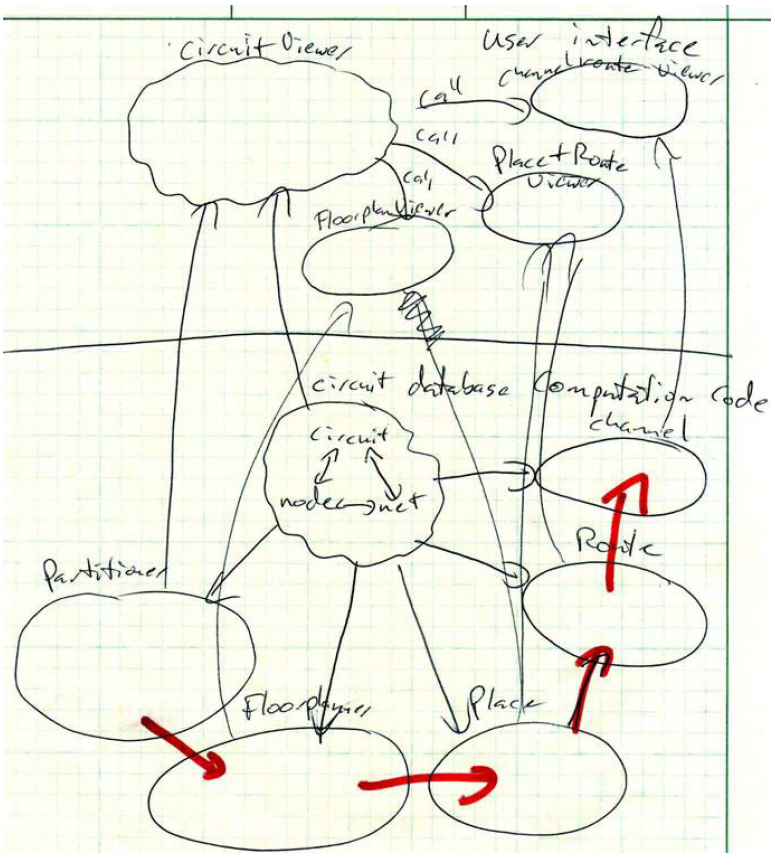


Figure 1: A runtime architecture of an 8,000-line system drawn by an experienced programmer with a Ph.D. in computer science but no formal training in software architecture [6]. The drawing follows the Model-View design pattern with the user interface above the line in the middle of the diagram and the circuit and computational code below the line. Notice node and net inside circuit's sub-architecture. The unlabeled arrows (including the thick arrows) represent data flow while the arrows labeled *call* represent control flow.

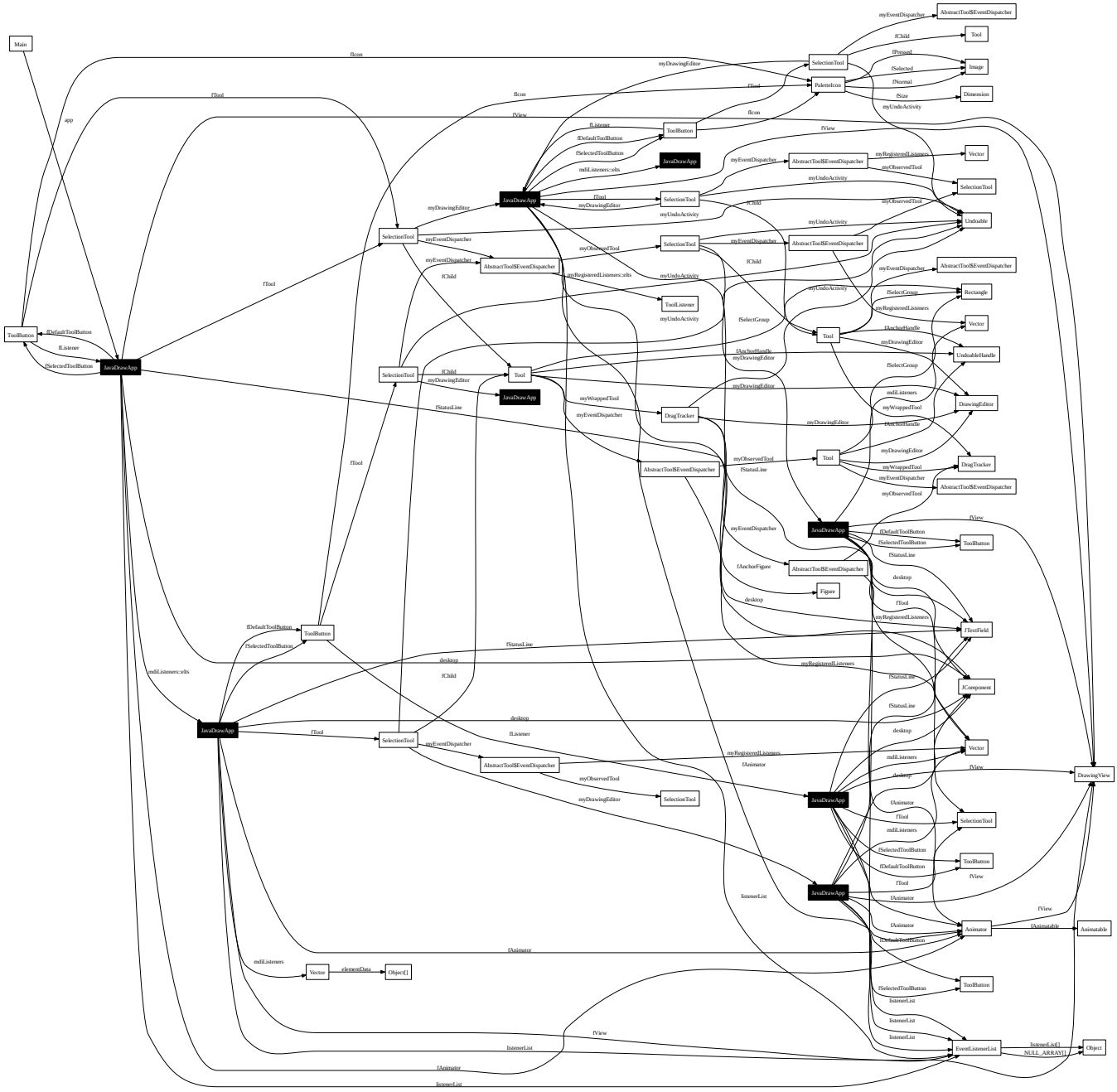


Figure 2: Thumbnail of the JHotDraw object graph obtained at compile-time by Womble [21]. The embedded image becomes readable after zooming in by 800%.

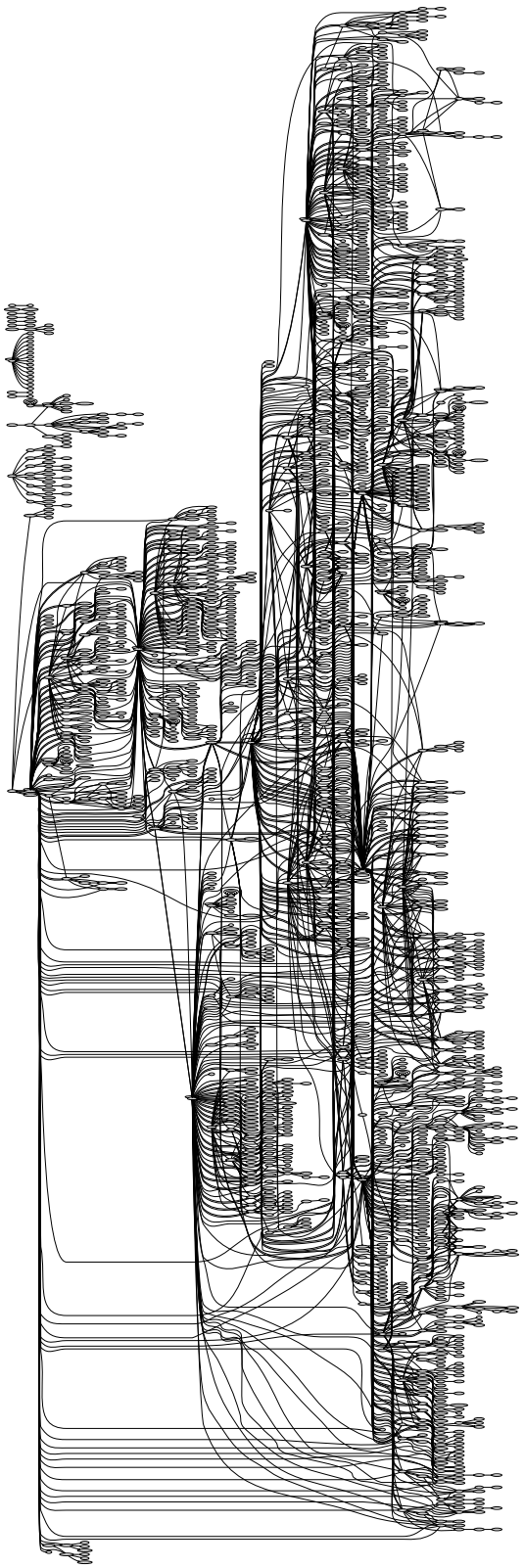


Figure 3: Flat object graph for JHotDraw obtained using PANGAEA [46]. The edges correspond to object references. The image is embedded postscript: to obtain a readable diagram, view this document electronically and zoom in by at least 2400%

2.1 Requirements on Solution

We listed earlier several requirements on a solution, namely that it be a static analysis and not require language extensions. Furthermore, an analysis for object-oriented code must also handle aliasing, recursion and inheritance. Key properties of a runtime architecture include:

- **Component = Objects.** An architecture must show *components* that correspond to *runtime* entities. For object-oriented systems, a component represents an object or a group of objects. A group of objects must be a meaningful abstraction, such as the `circuit` or `viewer` in Figure 1.
- **Connector = Object Relations.** An architecture has *connectors* that correspond to relations between runtime entities. For object-oriented systems, a connector represents a runtime interaction between some object in one component and some object in another component.
- **Tier = Group.** An architecture often groups conceptually related components into runtime *tiers* or partitions.
- **Hierarchy.** 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.
- **Summarization.** Different executions generate a different number of objects. Furthermore, the number of objects in the runtime object graph is unbounded. The architecture must be a finite representation of the runtime object graph. It is common practice to represent multiple objects at runtime with one canonical component.
- **Scalability.** Meaningful architectures would be most helpful for large systems. An architecture must scale, i.e., the size of top-level diagram should remain mostly constant as the program size increases arbitrarily.
- **Aliasing.** Ignoring aliasing may produce a misleading architecture. For instance, WOMBLE sometimes shows multiple nodes for the same runtime object. In Figure 2, there are multiple `JavaDrawApp` nodes highlighted in black. Figure 2 also shows a separate `DrawingEditor` instance when it is the same object as the `JavaDrawApp` instance at runtime (`JavaDrawApp` extends `DrawingEditor`). If two components are shown as distinct when they are the same, an architectural analysis may assign them different values for a key `trustLevel` property. As a result, the validity of such an analysis may be suspect at best. Some object graph analyses do not ignore aliasing but use unscalable whole-program analyses [33].
- **Soundness.** An architecture must be *sound* and represent all objects and relations between objects that may exist at runtime. We define soundness as:
  - **Component Soundness:** An architecture is sound if for every runtime object graph, there exists a map from objects to components, such that each runtime object o is mapped to exactly one component C in the architecture, i.e., the same runtime object must not map to multiple components in the architecture.
  - **Connector Soundness:** If there is a runtime connection between object o_1 and object o_2 in the runtime object graph, then there is a connector between components C_1 and C_2 corresponding to o_1 and o_2.
  - **Tier Soundness:** If object o is in a runtime tier d in the runtime object graph, then component C corresponding to o is in tier D in the architecture.
- **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 with one component that represents the entire system is sound but imprecise. This definition of precision can be refined 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  class List<ELTS T> {
2    ELTS T obj; // "Virtual field"
3    ...
4  }
5  interface Listener {
6  }
7  class BaseChart<M> // Declare domain parameter M
8        implements Listener {
9    domain OWNED; // Declare protected domain OWNED
10   // Declare reference to List object in OWNED
11   // Inner annotation M is for the list elements
12   OWNED List<M Listener> listeners;
13 }
14 class BarChart<M> extends BaseChart<M> {
15 }
16 class PieChart<M> extends BaseChart<M> {
17 }
18 class Model<V> implements Listener {
19    domain OWNED;
20    // Inner annotation V is for the list elements
21    OWNED List<V Listener> listeners;
22 }
23 class Main {
24   domain DOCUMENT, VIEW; // Top-level domains
25   // Bind domain parameter V to actual domain VIEW
26   DOCUMENT Model<VIEW> model;
27   VIEW BarChart<DOCUMENT> barChart;
28   VIEW PieChart<DOCUMENT> pieChart;
29 }
```

Figure 4: Document-View system with annotations.

2.2 Mapping from Source to High-Level Models

Consider a two-tiered Document-View architecture where `BarChart` and `PieChart` components are in a `VIEW` tier, and render a `Model` component in a `DOCUMENT` tier. All types implement a `Listener` interface to respond to change notifications. The code for this example is in Fig. 4, where the annotations are shown as underlined. We will explain the annotations in Section 2.4.

One way to view architectural extraction is that it maps code elements to elements in a high-level architectural model. Let us qualify a component by the tier that contains it using the :: symbol. Our approach does not do this, but let us hypothetically map the `BarChart` class from the code (on the left) to the `barChart` element in a `VIEW` tier in the architecture (on the right).

```
class BarChart to VIEW::barChart
class Model to DOCUMENT::model
```

A class is not a runtime entity. So the above map does not produce a runtime architecture. The above could indicate that all instances of the `BarChart` class map to a `barChart` component. But in an object-oriented system, there is usually more than one instance of any given class, and each instance can map to a different component in a runtime architecture. Instead of mapping a class or all of its instances, we need to map runtime objects. But a static analysis knows only about field or variable declarations in the program, which denote references to runtime objects. In the line below, we use `System.barChart` to denote a `barChart` field declared in class `System`, which points to an instance of the `BarChart` class at runtime. So we map:

```
object System.barChart to VIEW::barChart
object System.model to DOCUMENT::model
```

In Fig. 6(a), dashed boxes represent runtime tiers, solid-filled boxes represent objects and edges represent field references. At runtime, instances of `BarChart` and `Model` each contain an `List` object that holds listener objects. So, similarly, we map:

```
object BarChart.listeners to VIEW::listeners
object Model.listeners to DOCUMENT::listeners
```
In a runtime architecture, we model these `listeners` as *part of* `BarChart`, `PieChart` and `Model` objects, instead of showing them at the top-level. Conceptually, each view has a separate `listeners` object, and the `listeners` object of a `pieChart` is distinct from that of a `barChart`, and that of a `model`. So we map:
```
object BarChart.listeners to VIEW::barChart.OWNED::listeners
object PieChart.listeners to VIEW::pieChart.OWNED::listeners
```
Fig. 6(b) uses the nesting of boxes to indicate hierarchical containment. The thick dashed borders indicate that these `listeners` are *owned* or strictly encapsulated by their outer components.

A key difference between the code and the runtime architecture is that a single code element, e.g., the `Listener` element of a `List<Listener>`, could map to multiple design elements, based on the context. Indeed, `model` registers itself as a `Listener` for a `barChart`, and vice versa. So we map:
```
object Listener in BarChart.listeners to DOCUMENT::model
object Listener in Model.listeners to VIEW::barChart
```
To get this mapping, one solution is to annotate the `Listener` reference inside a `List<Listener>` by some parameter `ELTS`:
```
object Listener in List<Listener> to ELTS::obj
```
and to bind the `ELTS` parameter once to the `DOCUMENT` context, and once to the `VIEW` context [25]. Our system uses a similar solution. But in addition to being able to bind a context parameter to one of the top-level contexts, it can bind a context parameter to a local nested context, such as `barChart.OWNED` or `model.OWNED`. This expressiveness is crucial to extract a hierarchical representation.

Of course, multiple code elements could map to the same element in the runtime architecture. A reference of type `Model` and one of type `Listener`, an interface that `Model` implements, could alias and refer to the same object. So, they must map to the same component in the runtime architecture. Moreover, an analysis for object-oriented code must handle inheritance. In the above example, `PieChart` and `BarChart` extend a `BaseChart` class, and `BaseChart` declares the `listeners` object.

Finally, hierarchy enables displaying or eliding information at any level to show overviews of the runtime architecture at the desired level of abstraction. The (+) symbol indicates that the sub-structure of an object is elided (Fig. 6(c)).

In summary, this paper generalizes previous work [25], accounts for inheritance and aliasing, and extracts a hierarchical representation of the runtime structure of source code entities.

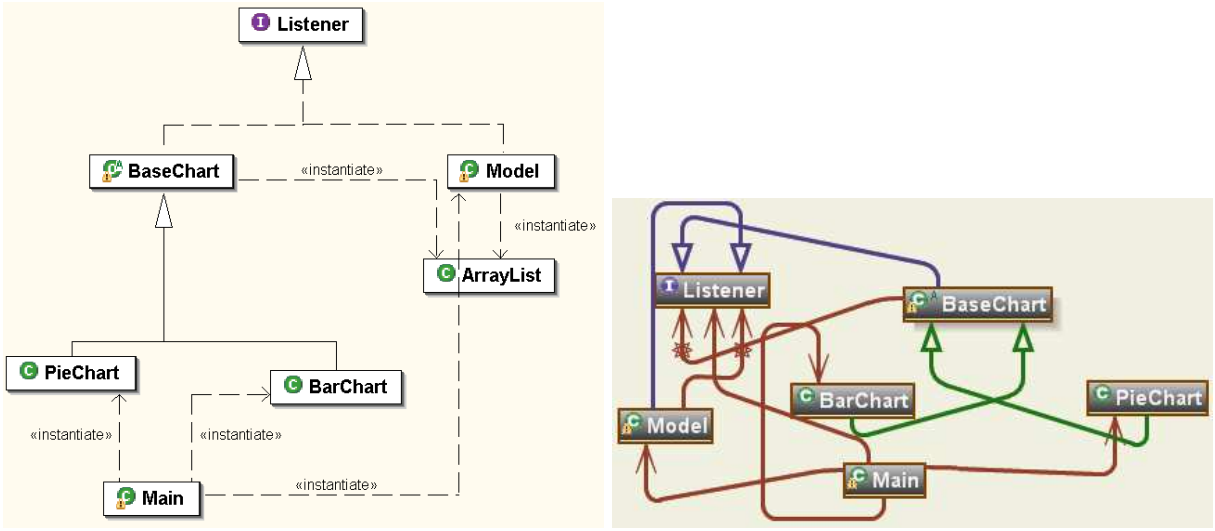
2.3 Strategy: Ownership Annotations

In this paper, we propose a principled two-pronged approach for extracting statically a runtime architecture of a system: (a) assume developers add to the source code annotations to clarify the architectural intent; and (b) use a sound static analysis that leverages the annotations and the code to extract a sound runtime architecture.

For adoptability, the annotations we propose to use are not radical language changes and do not affect the system's runtime semantics. The annotations support existing object-oriented languages, design idioms, frameworks and libraries. Instead of specifying components and connectors directly in code [6], the annotations specify and enforce the sharing of data between objects and constrain how the program can alias objects [5], which is a significant problem in creating architectural models.

In object-oriented programs, this state sharing is often not explicit but instead is implicit in the structure of references created at runtime. The idea of using annotations to recover a design from the code is not new [25]. But previous annotation-based systems did not specify the runtime instance structure or data sharing precisely and did not handle inheritance [25].

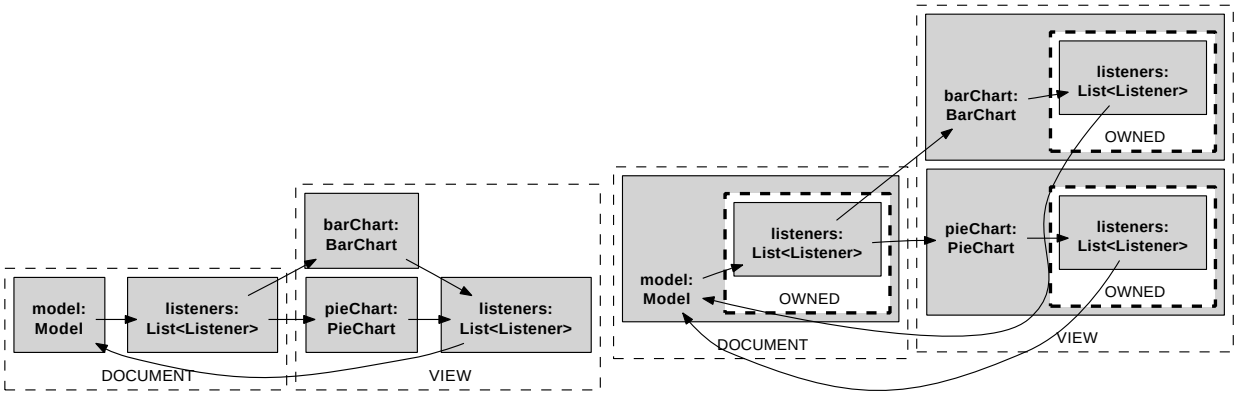
Ownership type annotations are appealing because they track *objects* not classes[4]and provide some precision about aliasing [12, 5]. Moreover, a type system checks the annotations at compile-time. Different ownership type systems have various degrees of expressiveness, but all support making an object *owned* by, i.e., part of another object's representation, to enforce *instance encapsulation*. This is a stronger guarantee than changing the visibility of a field by marking it `private`; the latter does not prevent a developer from



(a) Code architecture by Eclipse UML.

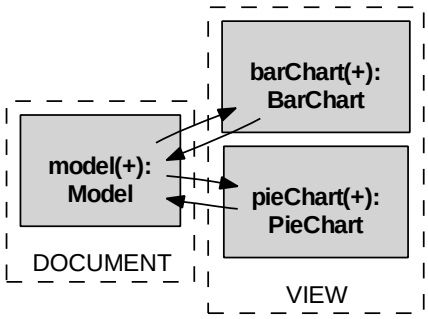
(b) Code architecture by AgileJ.

Figure 5: Code architecture for Document-View system.



(a) Flat runtime architecture.

(b) Hierarchical runtime architecture using nested boxes.



(c) Eliding the sub-structures.

Figure 6: Runtime architecture for Document-View system.

defining a `public` method that returns an alias.

2.4 Ownership Domains: Quick Overview

The ownership domains type system by Aldrich and Chambers uses annotations on the reference types in the program to make the state sharing between objects more explicit [5]. A developer indicates what domain an object is part of by annotating each reference to that object in the program.

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. Each object belongs to a single ownership domain that does not change at runtime.

Ownership domains may be declared at the top level of the application or within an object. Each object can declare one or more *public* or *private* domains to hold its internal objects, thus supporting hierarchy. Permission to access an object implies permission to access its public domains. Two objects can access objects in the same domain by declaring a formal *domain parameter* on one object and *binding* that formal domain parameter to another domain as long as the permissions allow that access. Finally, objects inside a private domain are encapsulated — unless a policy explicitly links a domain parameter to the private domain.

Relevance to Architectures. We propose a straightforward mapping between ownership domains and the architectural concepts we discussed earlier. Ownership domain annotations support abstract reasoning about data sharing by assigning each object in the runtime object graph to a single ownership domain.

The ownership domains declared at the top level map to the system's runtime tiers. Ownership domains declared within an object express a sub-architecture within the object, one that consists of other objects that represent its parts. This hierarchical containment relation enables architectural abstraction: the top-level domains may have only a few architecturally relevant objects, i.e., components. And each of those components can be made up of more objects representing subcomponents and so on, until low-level less architecturally relevant objects are reached. No programming language has an explicit tier construct, but ownership domain annotations can express and enforce a tiered architecture in code [5, 3].

The annotations also describe policies that govern references between ownership domains. Objects within the same ownership domain can refer to one another. But references can only cross domain boundaries if there is a *domain link* between the two domains [5]. Each object can declare a policy to describe the permitted aliasing among objects in its internal domains, and between its internal domains and external domains. ADLs typically express such policies using constraints.

In short, the annotations specify and enforce in code, architectural intent related to object encapsulation, logical containment (hierarchy), architectural tiers and object communication permissions.

Example. Figure 7 shows two classes with ownership domain annotations. In this paper, we use a simplified syntax similar to Java generics, but the concrete syntax uses existing language-support for annotations [3]. Domain names are arbitrary (except for a few special annotations); we use capital letters to distinguish them from other identifiers.

Class `DataAccess` declares a public domain `STATE`. and `Integer` and `Number` objects in `STATE`. Any object that has a reference to a `DataAccess` object can access the objects in its public domain. In addition, `DataAccess` requires some environment state that it does not own, so it declares a domain parameter `PENV`. The outer `PENV` annotation is for the list object itself; the inner `PENV` annotation is for the list elements, the `Integer` objects. Some other object of type `UnitTest`, which has the `ENV` domain, binds its `ENV` domain to `DataAccess`'s `PENV` domain parameter so that both objects can access the same state.

Next, we use the ownership domain annotations to extract statically a sound runtime architecture of a system.

[4]Related to ownership types, confined types track *classes* not objects [10].

```
1  class DataAccess<PENV> { // Declare domain parameter PENV
2    public domain STATE; // Declare public domain STATE
3    STATE Integer int1; // Declare Integer reference in STATE
4    STATE Number num1;
5
6    // Outer PENV annotation is for the ArrayList reference (Line 9)
7    // ArrayList has domain parameter ELTS for its elements
8    // Nested PENV annotation is bound to ArrayList ELTS (Line 9)
9    PENV ArrayList<PENV Integer> v2;
10 }
11 class UnitTest {
12   domain DATA, ENV; // Declare top-level domains
13
14   // Bind domain parameter PENV to actual domain ENV
15   DATA DataAccess<ENV> dataAccess;
16
17   static void main(lent String[shared] args) {
18     lent UnitTest test = new UnitTest();
19   }
20 }
```

`D C cObj`: declare object `cObj` of type `C` in domain `D`;
[**public**] **domain** `D`: declare private [or public] domain `D`;
`class C<P>`: declare formal domain parameter `P` on class `C`;
`C<ACTUAL> cObj`: bind formal domain parameter;
link `b -> d`: give domain `b` access to domain `d`;
Special Annotations. A few special annotations add expressiveness to the type system [5]:

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

Figure 7: `DataAccess` code with annotations.

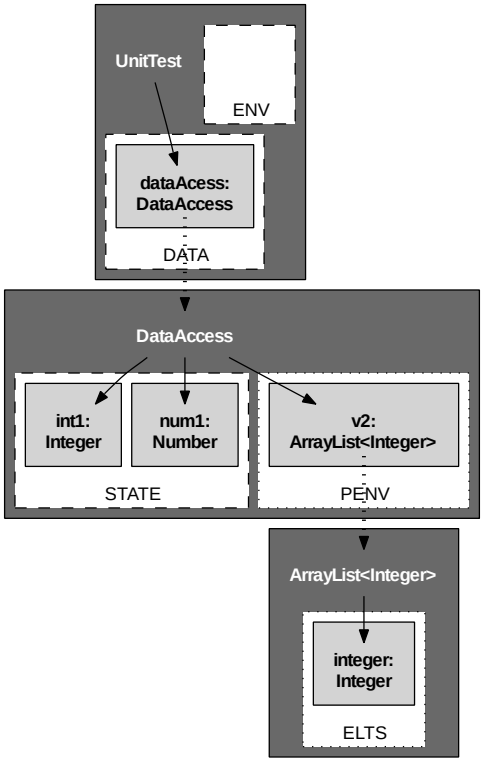


Figure 8: The *abstract graph* for `DataAccess`. A dark-filled box represents a type with white-filled domains declared inside it, and grey objects declared inside each domain. A formal domain parameter such as `PENV` has a dotted border. Actual ownership domains such as `STATE` and `ENV` have a dashed border. A thick dotted edge represents a type relationship (*is-a*). A solid edge represents a field reference (*has-a*).

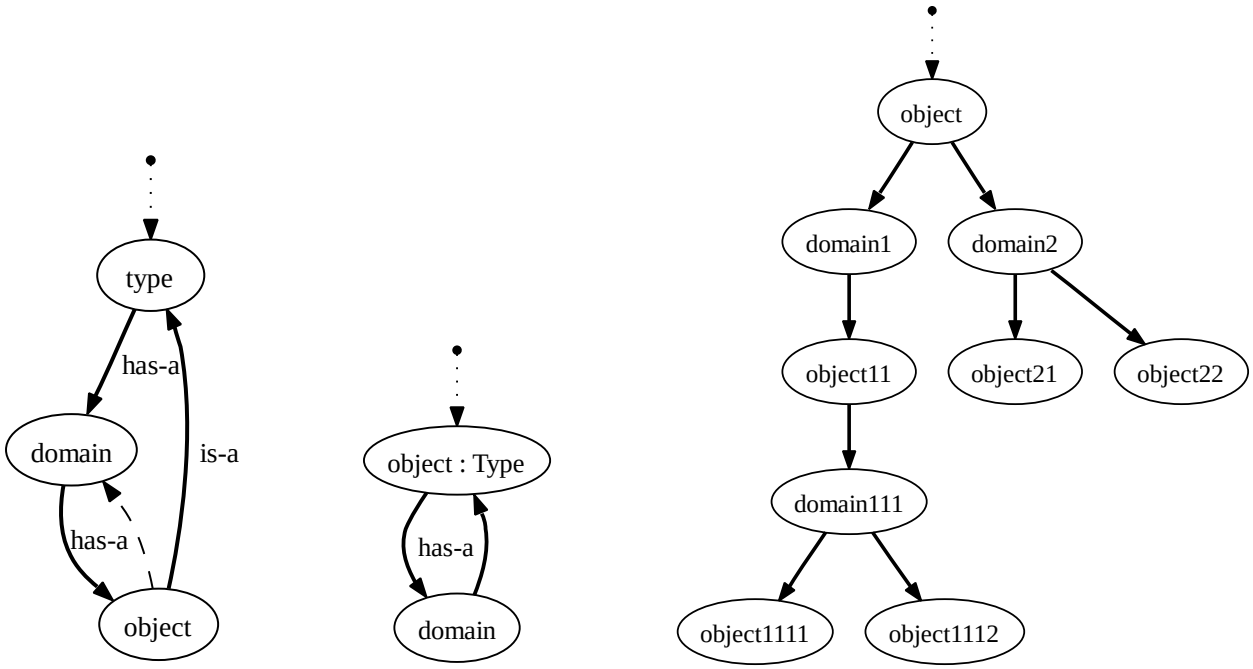


Figure 9: Graphical illustration of the *abstract graph*, *runtime graph* and *display graph*.

3 Analysis

The analysis uses different intermediate representations to extract the runtime architecture: it first builds an *abstract graph*, converts it into a *runtime graph*, from which it builds a *display graph*, also known as the Ownership Object Graph (OOG) (See Figure 9).

3.1 Abstract Graph

The *abstract graph* has the *abstract domains* declared in each *abstract type*. Each abstract domain represents fields and variables declared inside it as *abstract objects*. A visitor builds the *abstract graph* from the Abstract Syntax Tree (AST) of the annotated program (it must also account for certain characteristics discussed in Section 5). Figure 8 shows the abstract graph of the `DataAccess` system.

The abstract graph is inadequate as an architecture:

- **No instances.** The abstract graph is not hierarchical in the sense of an object having children. Rather, an abstract object has an abstract type, an abstract type has abstract domains, and an abstract domain has abstract objects. For example, abstract object `dataAccess` has type `DataAccess`, and abstract type `DataAccess` has domains `STATE` and `PENV`, and abstract domain `STATE` contains the abstract object `int1:Integer`.
- **Aliasing unaware.** The abstract graph does not reflect possible aliasing. The ownership domains type system guarantees that two objects in different domains can never alias, but two objects in the same domain may alias. As discussed in Section 2.1, if two objects could be aliased, the architecture must show them as one. In the example above, `int1:Integer` and `num1:Number` in the `STATE` domain may refer to the same object.
- **Incomplete.** An abstract domain in the abstract graph does not directly show all the objects that are

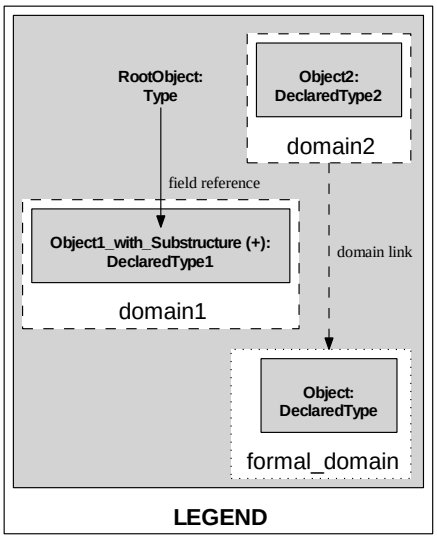


Figure 10: Legend for ownership domains.

in a given domain. It contains abstract objects only for the locally declared fields. E.g., `DataAccess` declares its `v2:ArrayList` field in its domain parameter `PENV`. Such non-local fields do not appear where the domain is declared. Hence, the empty `ENV` domain inside `UnitTest` in the abstract graph in Figure 8.

So the analysis converts the *abstract graph* into a *runtime graph* to approximate the true runtime object graph (ROG)[5].

We represent ownership domains as in Figure 10. A dashed border white-filled rectangle represents an actual ownership domain. A solid border grey-filled rectangle with a bold label represents an object. A dashed edge represents a link permission between two ownership domains. A solid edge represents a creation, usage, or reference relation between two objects. An object labeled "obj : T" indicates an object of type T as in UML object diagrams. The symbol (+) is appended to an object's label when its substructure is elided.

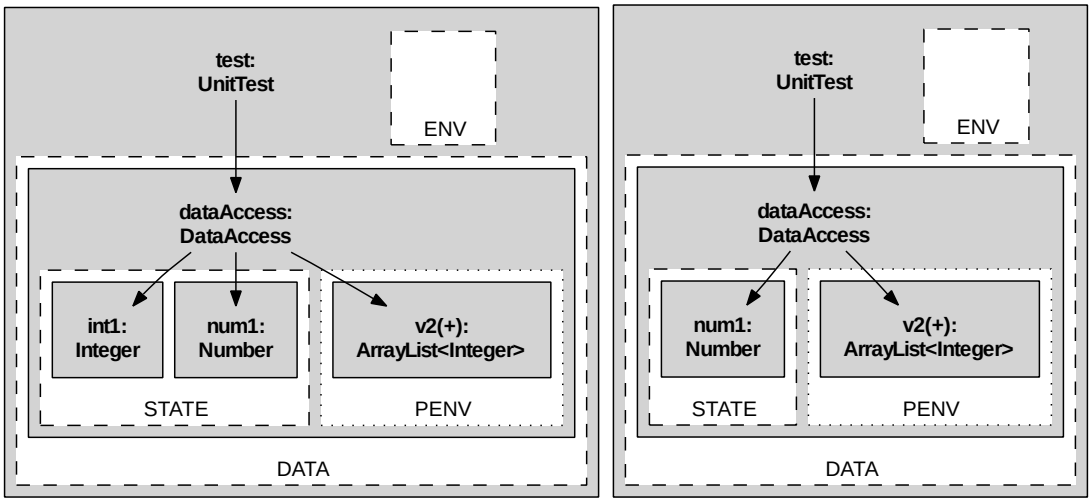
3.2 Runtime Graph

The *runtime graph* instantiates the types in the abstract graph and shows only objects and domains. Each *runtime object* contains *runtime domains* and each *runtime domain* contains *runtime objects*. Thus, in the runtime graph, one can view the children of an object without going through its declared type. Furthermore, to support the goals in Section 2.1, the runtime graph must address *object merging*, *object aliasing*, *object pulling* and *object edges*. We discuss each one in turn.

Object Merging. Different executions may generate a different number of objects, but an architecture must represent all possible executions. To address this, the runtime graph summarizes multiple runtime objects with a canonical runtime object. Further, exactly one object in the runtime graph represents each object in the ROG.

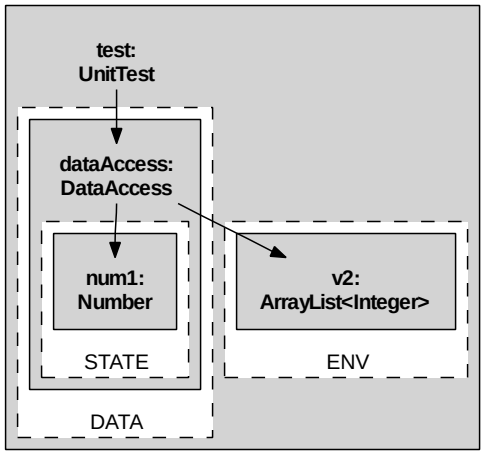
For instance, a dynamic analysis might display individual cells in a linked list of `Integer` objects, as `cons1:Cons`, `cons2:Cons`, etc. In our approach, all the `Cons` cells would get unified into a `cons1:Cons` object and a self-edge would represent the reference to the next cell, as shown in Figure 12(a).

[5]The runtime graph is an *approximation* of the true runtime object graph. We will explicitly refer to the true *runtime object graph* (ROG) to avoid confusion with the runtime graph. We hope the meaning will be clear from the context. A dynamic analysis take the runtime graph as input and does not have to compute it.



(a) *Runtime graph without* merging and *without* pulling.

(b) *Runtime graph with* merging but *before* pulling.



(c) *Runtime graph after* pulling.

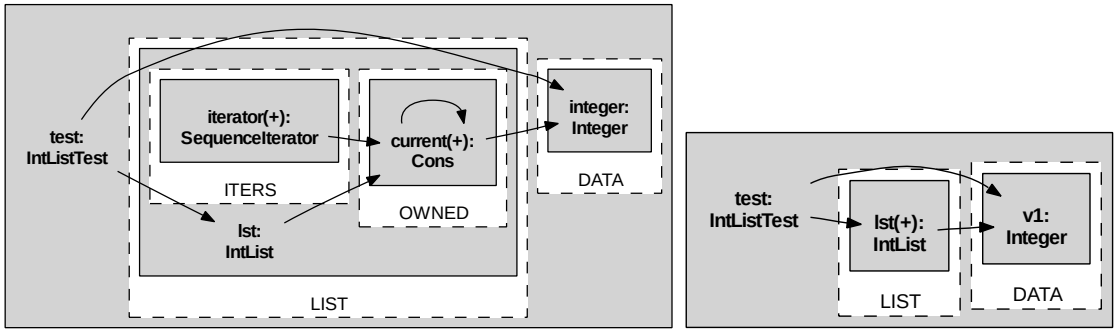
Figure 11: Partial `DataAccess` runtime graphs under construction.

Object Aliasing. When converting abstract objects from the abstract graph into runtime objects, the analysis merges two abstract objects *in the same domain*, if their types are related by inheritance. The ownership domains type system guarantees that two objects in different domains can never alias.

The runtime graph for the annotated code in Figure 7 is in Figure 11(b). One runtime object, labeled with `num1:Number`, merges the abstract objects `int1` and `num1` in domain `STATE` (`Integer` is a subtype of `Number`). The runtime graph still shows object `v2` in its formal domain parameter `PENV`.

Object Pulling. For soundness, each runtime object that is actually in a domain must appear in that domain in the runtime graph. To ensure this property, an abstract object declared inside a formal domain is *pulled* into each actual domain that is bound to the formal domain parameter.

Figure 11(b) shows object `v2` in the formal domain parameter `PENV` (dotted border). In Figure 11(c), object `v2` is pulled from the formal domain parameter `PENV` to the actual domain `ENV` in `UnitTest` (the former is bound to the latter using the annotation `DataAccess<ENV>` in Figure 7). In most cases, we elide formal domains after pulling, so Figure 11(c) no longer displays `PENV`. Similarly, an `ArrayList<Integer>` object has a domain parameter `ELTS` that contains `Integer` objects; those get pulled from `ELTS` into `ENV`



(a) OOG for an `IntList` object.

(b) Summary edge between objects `lst` and `v1`.

Figure 12: Summary edges added after limiting the projection depth or hiding `lst`'s substructure.

(See Figure 17).

Object Edges. Finally, the analysis adds field reference edges to the RuntimeGraph, shown as solid edges in Figure 11(c). Most edges are straightforward. For instance, `DataAccess` declares the two fields `int1` and `num1` in domain `STATE`. Objects `int1` and `num1` were merged so there is a field reference edge from a `DataAccess` object to the merged object. In future work, it is possible to add usage edges that show field accesses or method invocations.

3.3 Ownership Object Graph (OOG)

A runtime object can contain itself, so the runtime graph must represent a potentially unbounded ROG with a finite representation. For example, consider a class `QuadTree` that declares several fields of type `QuadTree` in its `owned` domain[6]:

```
class QuadTree {
  owned QuadTree _nwQuadTree;
  ...
}
```

Since there is a unique canonical object for each type in each domain, the object representing `QuadTree` in domain `owned` must also represent the child object of type `QuadTree` in the `owned` domain of the parent; it is therefore its own parent in this representation. A finite representation is essential to ensure that the analysis terminates. But, in a hierarchy, no object is its own parent. So the analysis creates the OOG as a finite depth-limited unrolling of the runtime graph. In the example above, we show one `QuadTree` object within another, down to a finite depth (`QuadTree` is the middle of Figure 27).

Cycle Detection. To break the recursion in the runtime graph, the analysis that generates the OOG stops when, from a given runtime object, it reaches the same runtime object a second time. Unlike the runtime graph, the OOG is a strictly hierarchical structure.

Edge Summaries. The OOG is depth-restricted but must still show all relations that exist at runtime. Merely truncating the recursion may fail to reveal all relations. For instance, child objects in a hierarchy may have fields that point to external objects, and the child objects may be beyond the visible depth. The analysis automatically adds summary edges from the parent objects to those external objects.

For example, consider a list of `Integer` objects. `IntList` declares a public domain `ITERS` for its iterators and a private domain `OWNED` to hold the linked list[7]. After pulling, the `head` of the list refers to a `Integer` object in a `DATA` domain containing the list elements (Figure 12(a)). If the projection depth is reduced to

[6] `owned` is a default private domain that need not be declared [5].

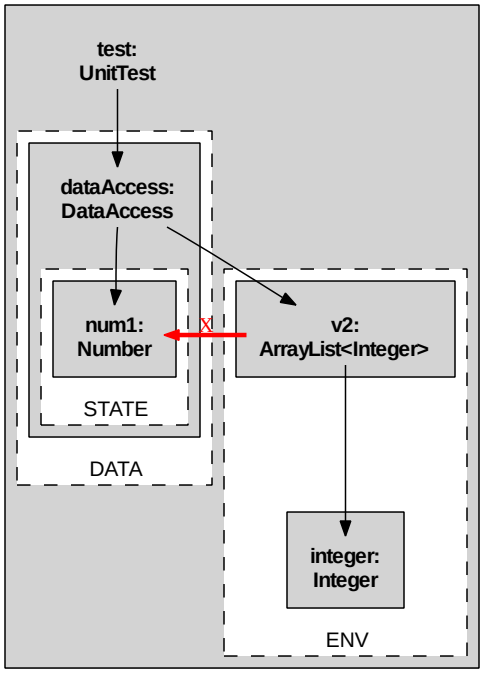


Figure 13: Final `DataAccess` OOG.

elide `IntList`'s substructure, the analysis adds a summary edge from `IntList` to the pulled `Integer` object in Figure 12(b).

Figure 13 shows the final `dataAccess` OOG. In addition to the domains and objects previously discussed, the `ENV` domain declares an object `Integer`. Object `Integer` was pulled from the `ELTS` formal domain parameter to the `ENV` domain. The bold edge marked with the symbol `X` is an example of an edge that is not added, even though `Integer` is a subtype of `Number` because the `ELTS` formal domain parameter is not bound to the `STATE` domain. This shows how OOG edges are more precise than those obtained by superimposing field reference edges based on the associations in a class diagram.

Recapitulation. The 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 domain. There are two edge types. Edges between objects correspond to field reference or usage relations. Edges between domains correspond to domain links. Finally, the OOG handles object merging, object aliasing and object pulling. Compared to earlier definitions of object graphs [36], the OOG explicitly represents clusters of objects (domains) and edges between these clusters (domain links). In contrast to other object ownership hierarchies [20, 35], in an OOG, the owner of an object is a domain not another object.

Finally, in this paper, we sometimes elide the root domain and root object from the displayed OOGs for readability. Often times, the root object is an instance of a fake class whose sole purpose is to declare the top-level domains [3].

4 Formalization

In this section, we formally describe the analysis as a rewrite system to generate instances from types, merge equivalent instances in a domain, and deduce edges between instances.

[7]This is a canonical example to evaluate an ownership type system's expressiveness. The annotated code is in the ownership domains paper [5].

4.1 Rewriting Rules

We use a labeled record notation for the data type declarations of the AbstractGraph and the RuntimeGraph. We use $(\ldots)$ for a tuple, $\{o\ldots\}$ for a set and $[d\ldots]$ for a sequence. We use $<:$ to denote subtyping. We sometimes qualify a domain d by the type T that declares it as $T{::}d$. We describe the algorithm to construct a RuntimeGraph from an AbstractGraph using small-step rewriting rules (Fig. 14).

To help keep the representations distinct, we use English letters $(o, d, \ldots)$ for AbstractGraph elements and Greek letters $(\theta, \ldots)$ for RuntimeGraph elements. The AbstractGraph consists of the AbstractTypes in the program, the AbstractDomains declared in each type and the AbstractObjects declared in each domain. Each AbstractObject maintains bindings, each from a formal to an actual domain, shown as $(d_{formal} \mapsto d_{actual})$ to avoid ambiguity.

There are no RuntimeDomains. To avoid copying, we directly add AbstractDomains to the RuntimeGraph. A RuntimeObject knows what AbstractDomain owns it and maintains a set of AbstractObjects it merges. Since all the AbstractDomains are represented in the RuntimeGraph, the analysis converts AbstractObjects into RuntimeObjects, starting with a root AbstractObject.

A context Θ is the set of valid RuntimeObjects that are part of the RuntimeGraph. Once a RuntimeObject is removed from Θ, as may happen during a replace operation, it is no longer part of the RuntimeGraph.

Given the list of all RuntimeObjects $(\{o_i\ldots\},\ d)$ in Θ, the RuntimeObjects that are in a given AbstractDomain d_x, are those in Θ that have $d = d_x$.

The analysis obtains the AbstractDomains inside a RuntimeObject θ by looking up each AbstractObject $o_i : T_i$ that θ merges, the declared AbstractType T_i of each o_i, and each AbstractDomain d_i that T_i declares.

The algorithm works by applying these rules until it can no longer generate new facts, i.e., RuntimeObjects and RuntimeEdges. Some rules add RuntimeObjects to the context Θ, other rules replace existing RuntimeObjects with others. Despite this non-monotonicity, the algorithm is stable because rule preconditions prevent regenerating facts that have been removed.

For a given input, we believe the rules will always produce the same graph structure, regardless of the potentially non-deterministic order in which the rules are applied. A different execution of the rules may, however, label a RuntimeGraph differently. A RuntimeObject merges multiple AbstractObjects and each AbstractObject might have multiple types. The analysis picks one of those types nondeterministically as the label for a RuntimeObject.

Subtyping and Type Compatibility. R-Aux-Compat defines type compatibility: the first two disjuncts are necessary to handle the potential aliasing of variables based on subtyping, the third and fourth disjuncts are heuristics which we discuss in Section 5 and can be turned off. The rules use ownership domains subtyping [5], which follows standard nominal subtyping, and in addition, checks that all domain parameters are invariant with subtyping.

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

Runtime Objects. The judgment for creating objects is of the form $\Theta \implies \Theta'$. Before creating a RuntimeObject for an AbstractObject o of type t in AbstractDomain d, the analysis checks if d already has a RuntimeObject θ of type t' where t and t' are compatible according to R-Aux-Compat. If such an object does not exist, R-New-Object creates a new RuntimeObject, which we represent as $\theta = (\{o\ldots\}, d)$. If there exists a RuntimeObject $\theta = (\{o_1\ldots\}, d)$, R-Merge-Objects replaces θ with a new RuntimeObject that unions the two sets $\{o\ldots\}$ and $\{o_1\ldots\}$.

An object about to be created in a domain may have a type that is compatible with two existing RuntimeObjects that are not compatible with each other. In this case, the new object merges nondeterministically with one of the existing objects, and then merges with the other using R-Merge-Existing. This avoids multiple interface inheritance from triggering unsoundness [2].

Next, R-Pull-Object pulls up each RuntimeObject θ from its owning formal domain d_{param} into the d_{actual} domain bound to d_{param}, possibly replacing RuntimeObjects (See Fig. 15 for an illustration of the rule).

$$
\begin{array}{lll}
h \in \mathsf{AbstractGraph} & ::= & (\mathbf{RootObject} = o,\ \mathbf{AbstractTypes} = \{t \ldots\}) \\
t \in \mathsf{AbstractType} & ::= & (\mathbf{Id} = t,\ \mathbf{Domains} = \{d \ldots\},\ \mathbf{Edges} = \{e \ldots\}) \\
d \in \mathsf{AbstractDomain} & ::= & (\mathbf{Id} = d,\ \mathbf{Objects} = \{o \ldots\},\ \mathbf{DeclaringType} = t) \\
o \in \mathsf{AbstractObject} & ::= & (\mathbf{Id} = o,\ \mathbf{Domain} = d,\ \mathbf{DeclaredType} = t,\ \mathbf{Bindings} = \{b \ldots\}) \\
b \in \mathsf{Binding} & ::= & (\mathbf{FormalDomain} = d_{Formal} \mapsto \mathbf{ActualDomain} = d_{Actual}) \\
e \in \mathsf{AbstractEdge} & ::= & (\mathbf{FromType} = t_{src},\ \mathbf{ToDomain} = d_{dst},\ \mathbf{ToType} = t_{dst}) \\
\gamma \in \mathsf{RuntimeGraph} & ::= & (\mathbf{RootObject} = \theta, \mathbf{Objects} = \{\theta \ldots\},\ \mathbf{Edges} = \{\eta \ldots\}) \\
\theta \in \mathsf{RuntimeObject} & ::= & (\mathbf{MergedObjects} = \{o \ldots\},\ \mathbf{OwnerDomain} = d) \\
\eta \in \mathsf{RuntimeEdge} & ::= & (\mathbf{FromPath} = [d_{src} \ldots],\ \mathbf{FromType} = t_{src},\ \mathbf{ToPath} = [d_{dst} \ldots],\ \mathbf{ToType} = t_{dst}) \\
\Theta & ::= & \emptyset \mid \Theta, \mathsf{RuntimeObject}(\{o \ldots\}, d)
\end{array}
$$

$$\frac{\mathsf{AbstractObject}(o,\ d,\ t,\ \{b \ldots\})}{\Theta \vdash \mathsf{try}(\{o\},\ d)}[\text{R-CONVERT-OBJECT}]$$

$$\frac{\begin{array}{c}\mathsf{RuntimeObject}(\{o_{pull} \ldots\},\ d_{param}) \in \Theta \qquad \mathsf{RuntimeObject}(\{o_{parent} \ldots\},\ d_{parent}) \in \Theta \\ \mathsf{AbstractDomain}(d_{param}, \mathsf{typeof}(o_{parent})) \qquad \mathsf{aparam}(o_{parent}, d_{param}, d_{actual})\end{array}}{\Theta \vdash \mathsf{try}(\{o_{pull} \ldots\},\ d_{actual})}[\text{R-PULL-OBJECT}]$$

$$\frac{\Theta \vdash \mathsf{try}(\{o \ldots\},\ d) \qquad \forall o_1, \forall \mathsf{RuntimeObject}(\{o_1 \ldots\},\ d) \in \Theta \qquad \nvdash \mathsf{compat}(\mathsf{typeof}(o),\ \mathsf{typeof}(o_1))}{\Theta \implies \Theta, \mathsf{RuntimeObject}(\{o \ldots\},\ d)}[\text{R-NEW-OBJECT}]$$

$$\frac{\Theta \vdash \mathsf{try}(\{o \ldots\},\ d) \qquad \mathsf{compat}(\mathsf{typeof}(o),\ \mathsf{typeof}(o_1))}{\Theta, \mathsf{RuntimeObject}(\{o_1 \ldots\},\ d) \implies \Theta, \mathsf{RuntimeObject}(\{o \ldots\} \cup \{o_1 \ldots\},\ d)}[\text{R-MERGE-OBJECTS}]$$

$$\frac{\mathsf{compat}(\mathsf{typeof}(o_1),\ \mathsf{typeof}(o_2))}{\Theta, \mathsf{RuntimeObject}(\{o_1 \ldots\},\ d), \mathsf{RuntimeObject}(\{o_2 \ldots\},\ d) \implies \Theta, \mathsf{RuntimeObject}(\{o_1 \ldots\} \cup \{o_2 \ldots\},d)}[\text{R-MERGE-EXISTING}]$$

$$\frac{\mathsf{AbstractObject}(o,\ d,\ t,\ \{d_{param} \mapsto d_{actual}, \ldots\})}{\mathsf{aparam}(o, d_{param}, d_{actual})} \qquad \frac{\mathsf{AbstractObject}(o,\ d,\ t,\ \{b \ldots\})}{\mathsf{typeof}(o) = t}$$

$\mathsf{compat}(t_1,\ t_2)$ iff $t_1 <: t_2$ or $t_2 <: t_1$ or $\mathsf{existsNonTrivialLUB}(t_1,\ t_2)$ or $\mathsf{mapToSameDIT}(t_1,\ t_2)$ [R-AUX-COMPAT]

$$\frac{\mathsf{RuntimeObject}(\{o...\}, d) \in \Theta \qquad \mathsf{AbstractObject}(o, d, t_{src}, \{b \ldots\}) \qquad \mathsf{AbstractEdge}(t_{src}, d_{dst}, t_{dst})}{\Theta;\ \mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{RuntimeEdge}([d], t_{src}, [d, d_{dst}], t_{dst})}[\text{R-NEW-EDGE}]$$

$$\frac{\begin{array}{c}\Theta;\ \mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{RuntimeEdge}([d_{src}...], t_{src}, [d_{dst}...], t_{dst}) \qquad \mathsf{AbstractDomain}(d, \mathsf{typeof}(o_{parent})) \\ \mathsf{RuntimeObject}(\{o_{parent} \ldots\},\ d_{parent}) \in \Theta \qquad \mathsf{nocycle}([d_{parent}, d_{src}...]) \qquad \mathsf{nocycle}([d_{parent}, d_{dst}...])\end{array}}{\Theta;\ \mathsf{RuntimeObject}(\{o_{parent}...\}, d_{parent}) \vdash \mathsf{RuntimeEdge}([d_{parent}, d_{src}...], t_{src}, [d_{parent}, d_{dst}...], t_{dst})}[\text{R-PULL-EDGE}]$$

$$\frac{\begin{array}{c}\Theta;\ \mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{RuntimeEdge}([d_{src_1}, d_{src_2}...], t_{src}, [d_{dst}...], t_{dst}) \\ \mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{mapFtoA}(d_{src_1}, d_{src_2}) = d_{src'}\end{array}}{\Theta;\ \mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{RuntimeEdge}([d_{src'}...], t_{src}, [d_{dst}...], t_{dst})}[\text{R-SUBST-EDGE-L}]$$

$$\frac{\begin{array}{c}\Theta;\ \mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{RuntimeEdge}([d_{src}...], t_{src}, [d_{dst_1}, d_{dst_2}...], t_{dst}) \\ \mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{mapFtoA}(d_{dst_1}, d_{dst_2}) = d_{dst'}\end{array}}{\Theta;\ \mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{RuntimeEdge}([d_{src}...], t_{src}, [d_{dst'}...], t_{dst})}[\text{R-SUBST-EDGE-R}]$$

$$\frac{\mathsf{AbstractObject}(o, d, t, \{d_{formal} \mapsto d_{actual}, \ldots\})}{\mathsf{RuntimeObject}(\{o...\}, d) \vdash \mathsf{mapFtoA}(d, d_{formal}) = d_{actual}}[\text{R-PATH-SUBST}]$$

Figure 14: Data type declarations and rewriting rules to convert an AbstractGraph into a RuntimeGraph.

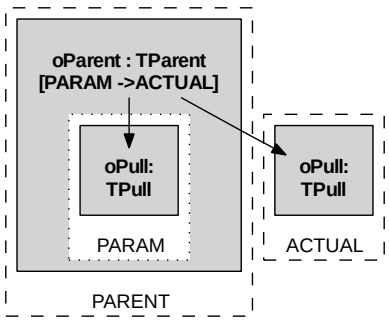


Figure 15: Illustration of the pulling rule: o_{pull} gets pulled from $PARAM$ to $ACTUAL$. Of course, $ACTUAL$ can be the same as $PARENT$.

Runtime Edges. An AbstractEdge comes from a field reference, and thus includes t_{src}, the type that declares the field, and d_{dst} and t_{dst}, the domain and type of the object the reference points to. Because RuntimeObjects may get replaced, a RuntimeEdge is not defined in terms of a source and destination RuntimeObjects. Instead, we define a RuntimeEdge as a source path p_{src}, a source type t_{src}, a target path p_{dest}, and a target type t_{dst}.

A path is a sequence, possibly empty, of type-qualified domains to traverse to locate an object. In Fig. 17, the root object is in a `lent` domain, which is a global domain. So the path $[d_{lent}, d_{DATA}]$ and the type $t_{\texttt{DataAccess}}$ uniquely identify the `dataAccess` object, reachable starting from the `lent` domain, and into the `DATA` domain in the `UnitTest` class (the tuples are in Fig. 18).

The judgment for creating RuntimeEdges is of the form:

$$\Theta; \mathsf{RuntimeObject} \vdash \mathsf{RuntimeEdge}(p_{src}, t_{src}, p_{dest}, t_{dst})$$

R-New-Edge converts an AbstractEdge into a RuntimeEdge by identifying an object o of type t_{src}, and recording the edge as two paths starting at the parent of o. The source path is the domain d of o. The destination path includes d followed by d_{dst}, the domain pointed to by the AbstractEdge. The destination type t_{dst} is the type of the field pointed to by the AbstractEdge.

A domain in the source or destination path of a RuntimeEdge might be a formal domain. In that case, R-Pull-Edge performs an edge pulling operation similar to pulling objects, so that edges appear in any actual domains to which the formal domain is bound. R-Pull-Edge may lengthen the domain paths. The rule checks that it does not create cyclic paths, to avoid non-termination, using the nocycle side condition.

R-Subst-Edge-L and R-Subst-Edge-R substitute formals with actuals in the source path and destination path, respectively, based on the binding information in the origin RuntimeObject. R-Path-Subst actually performs the substitution, by looking at the binding information in the origin RuntimeObject. In some cases, a substitution can shorten a path (see Fig. 18 for examples).

Edges and path-dependent domains. The ownership domains type system allows path-dependent annotations that are of the form `obj1.obj2...DOMAIN`, where `obj1`, `obj2`, ..., are chains of final fields or variables, and `DOMAIN` is a public domain declared on the type of the last object in the path. Handling these domains requires extending the formal system to define a Binding to be from a formal AbstractDomain to a sequence of AbstractDomains, instead of a single AbstractDomain, to generate longer paths for RuntimeEdges. We did not include this feature into the core formal system for simplicity, but it is straightforward to extend the data structures and the rules.

Domain Links. The analysis adds to the runtime graph RuntimeLinks in a similar manner to RuntimeEdges. Observe, in Figure 14, a RuntimeLink is a simpler RuntimeEdge, one that only has source and destination domain paths without source and destination types. Those are already computed when computing RuntimeEdges. Moreover, the underlying ownership domains type system guarantees that objects communicate only when permitted, so a RuntimeEdge cannot violate a RuntimeLink.

$$\frac{\begin{array}{l}\textit{T-Store}\\ \quad\ldots \qquad S[\ell] = C\!<\!\overline{\ell'.n}\!>\!(\overline{v}) \iff \Sigma[\ell] = C\!<\!\overline{\ell'.n}\!> \\ \mathit{fields}(\Sigma[\ell]) = \overline{T}\ \overline{f} \implies (S[\ell, i] = \ell'') \wedge (\Sigma[\ell''] <: T_i) \qquad \ldots\end{array}}{\Sigma \vdash S}$$

Figure 16: Partial store typing rule from FDJ [5].

4.2 Soundness

For the OOG to be most useful, it should be a *sound* approximation of the true runtime object graph for any program run. We formally revisit the earlier definitions in Section 2.1 and formally prove a key property. Since the OOG is just a depth-limited projection that should preserve the soundness of the runtime graph by adding summary edges, the true runtime object graph (ROG) relates to the RuntimeGraph as follows:

- **Unique Representatives:** Each object ℓ in the ROG is represented by exactly one representative in the RuntimeGraph. Similarly, each domain in the ROG is represented by exactly one domain in the RuntimeGraph. Furthermore, if object ℓ is owned by domain d in the ROG, then the representative of ℓ is owned by the representative of e in the RuntimeGraph. Similarly, if ℓ has a domain d in the ROG, then the representative for ℓ has a representative domain for d in the RuntimeGraph.
- **Edge Soundness:** If there is a field reference from object ℓ_1 to object ℓ_2 in the ROG, then there is a field reference edge between RuntimeObjects θ_1 and θ_2 corresponding to ℓ_1 and ℓ_2 in the RuntimeGraph, and similarly for domain links;

Soundness Proof. The proof builds on the formalization of ownership domains using Featherweight Domain Java (FDJ) [5]. Figure 16 shows a subset of the store typing rule *T-Store*. An overbar represents a sequence. In FDJ, locations represent object identity. 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. A type in FDJ is a class name and a set of actual ownership domain parameters. $S[\ell]$ denotes the store entry for ℓ. Given an object in the runtime object graph represented by location ℓ, $\Sigma[\ell] = C\!<\!\overline{\ell'.n}\!>$. Here, each $\ell'_i.n_i$ refers to a domain named n_i that is part of the runtime object ℓ'_i. By Rule *Aux-Owner*, the first actual domain is the owner, i.e., $\mathit{owner}(C\!<\!\overline{\ell'.n}\!>) = \ell'_1.n_1$[8]. $S[\ell, i]$ denotes the value in the ith field of $S[\ell]$. *T-Store* ensures that the store type Σ gives a type to each location in S, one that is consistent with the classes and actual ownership domain parameters in S. CT is the class table.

Proof of Unique Object Representatives. The proof is by induction over the ownership tree (from FDJ). The ownership relation has no cycles and is well-founded. The base case for the induction is trivial. The top-level object in the runtime object graph has a unique representative in the RuntimeGraph corresponding to the root RuntimeObject. We strengthen the inductive hypothesis (i.h.) as follows: *Each object in the runtime object graph of runtime type C is represented by exactly one RuntimeObject θ that merges an AbstractObject o of type C in the RuntimeGraph.*

We first show that the representative is *unique*. Given an object in the runtime object graph represented by location ℓ, $\Sigma[\ell] = C\!<\!\ell_o.n_1, \ldots\!>$. Similarly, $\Sigma[\ell_o] = T_o\!<\!\overline{d}\!>$. We also have $n_1 \in \mathit{domains}(T_o)$, i.e., n_1 is declared on T_o or a supertype thereof R_i with $T_o <: R_i$. By (i.h.), ℓ_o maps to a unique RuntimeObject θ_o that merges at least one AbstractObject of type T_o.

The RuntimeGraph obtains the AbstractDomains inside RuntimeObject θ_o from the declared type t of each AbstractObject that θ_o merges and the AbstractDomains that t declares. By (i.h.), θ_o merges an AbstractObject o such that $o : T_o$. By Rule R-Aux-Compat, there are two cases to consider:

Case $o_i : R_i$ and $T_o <: R_i$, for any such R_i. So either $T_o{::}n_1$ or $R_i{::}n_1$. For each domain declaration on a type, there is a unique AbstractDomain in the RuntimeGraph. Because AbstractDomains corresponding to inherited domains are unified, there is a unique AbstractDomain for $T_o{::}n_1$ and $R_i{::}n_1$, for any such R_i. This is ensured by the construction of the AbstractGraph (See Figure 19).

[8]The formal system treats the first domain parameter of a class as its owning domain. The practical system uses a slightly different syntax to emphasize the semantic difference between the owner domain of an object and its domain parameters.

Case $o_i : U_i$ and $U_i <: T_o$. AbstractDomain $(U_i{::}n_1)$ is the same as AbstractDomain $(T_o{::}n_1)$ and has the same representative in the RuntimeGraph for any such U_i.

Once AbstractDomain n_1 is uniquely identified, there can be only one RuntimeObject θ_o that merges $o : T_o$ inside n_1, by the *Unique Object per Domain and Type Lemma*.

Next, we show that the representative *exists*. Given an object in the runtime object graph represented by location ℓ, $\Sigma[\ell] = C{<}\ell_o.n_1, \ldots{>}$. In turn, ℓ_o is such that $\Sigma[\ell_o] = T_o{<}\ldots{>}$. ℓ must have been created by object ℓ_{this}. As before, $\Sigma[\ell_{this}] = T_{this}{<}\ell'_{this}.n', \ldots{>}$.

By rule *R-New* in FDJ, there must be a `new` $C{<}\overline{d}{>}(\overline{v})$ creation expression. Recall, FDJ treats the first domain parameter of a class as its owning domain. There are several cases depending on whether d_1 is of the form $n.x$ or a formal domain parameter.

Case d_1 is of the form $n.x$ where $\ell_{this} = \ell_o$. Then $T_o = T_{this}$. Then $n_1 \in \mathit{domains}(T_{this})$. By construction, all the AbstractDomains declared in the program exist in the RuntimeGraph, hence AbstractDomain n_1 exists. From the AbstractObject, R-New-Object will create a RuntimeObject that merges at least AbstractObject $o : C$ in n_1. So the representative exists.

Case d_1 is of the form $n.x$ where $\ell_{this} \neq \ell_o$. By induction, ℓ_o has a representative $(\{o' : T_o\}, n')$. Then $n_1 \in \mathit{domains}(T_o)$. The RuntimeGraph obtains the AbstractDomains inside from $o' : T_o$, the declared AbstractType T_o and AbstractDomain n_1 that T_o declares. Similarly to the case above, the RuntimeObject exists.

Case d_1 is a formal domain. By the *Binding Chains Lemma*, there exists a set of object creation expressions `new` $C{<}\overline{d}{>}(\overline{v})$ and a chain of formal to actual domain bindings. By the transitivity of the binding relation, we have $C ::\alpha_1 \mapsto^* T_o{::}n_1$. All the AbstractDomains declared in the program exist in the RuntimeGraph, hence AbstractDomain α_1 exists. By the AbstractGraph construction, there must exist an AbstractObject $o : C$ declared in α_1. By the *Object Pulling Lemma*, if o is merged by RuntimeObject θ_i in AbstractDomain d_{fi} and d_{fi} is transitively bound to d_{fj}, then o is merged by some RuntimeObject θ_j in AbstractDomain d_{fj}. Intuitively, the lemma means that the pulling of o either creates a new RuntimeObject owned by AbstractDomain d_{fj} or merges o into an existing RuntimeObject owned by d_{fj}. Thus, the representative RuntimeObject$(\{o : C \ldots\},\ n_1)$ exists.

The proof required the following lemmas which are well-formedness rules on the RuntimeGraph.

Lemma: Unique Object per Domain and Type. If there exists a RuntimeObject $\theta = (\{o : T, \ldots\}, d)$ and a RuntimeObject $\theta' = (\{o' : T', \ldots\}, d)$ with $T' <: T$ or $T <: T'$, then θ is the same as θ'.
Proof. Immediate from R-Merge-Objects. Note, this proof does not reflect the existence of virtual abstract objects. Although those virtual abstract objects appear in the RuntimeGraph (and violate the uniqueness invariant), they are omitted from the OOG.

Assume there are two such RuntimeObjects that merge AbstractObjects o and o' of types T and T' inside domain d, i.e., $\theta_1{=}(\{o : T, \ldots\}, d)$ and $\theta_2 = (\{o' : T', \ldots\}, d)$ with $T <: T'$ or $T' <: T$. By R-Aux-Compat, isCompat(T, T') true. Then R-Merge-Objects replaces θ_1 and θ_2 with $\theta = (\{o, o', \ldots\}, d)$.

Lemma: Object Pulling. If $\exists$ RuntimeObject$(\{o \ldots\}, d_{fi})$ and $d_{fi} \mapsto^* d_{fj}$ then $\exists$ RuntimeObject$(\{o \ldots\}, d_{fj})$.
Proof. By induction on the length of the binding sequence. We abbreviate RuntimeObject and AbstractDomain as RO and AD.

Base case $d_{fi} = d_{fj}$.

$\exists$ RO$(\{o \ldots\}, d_{fi})$	Assumption
$d_{fi} \mapsto^* d_{fi}$	Trivially
$\exists$ RO$(\{o \ldots\}, d_{fj})$	Since $d_{fi} = d_{fj}$

□

Assume: $\exists$ $RO(\{o\ldots\}, d_{fi})$ *and* $d_{fi} \mapsto^* d_{fj}$ *and* $d_{fj} \mapsto d_{fk}$. To show: $\exists$ $\mathsf{RO}(\{o\ldots\}, d_{fk})$

$\exists$ $\mathsf{RO}(\{o\ldots\}, d_{fi})$	Assumption
$d_{fi} \mapsto^* d_{fj}$	Assumption
$\exists$ $\mathsf{RO}(\{o\ldots\}, d_{fj})$	By i.h.
$d_{fj} \mapsto d_{fk}$	Assumption
$\exists$ $\mathsf{AD}(d_{fj}, T_{this})$	d_{fj} formal domain
$\exists$ $\mathsf{AO}(o_p,\ d_p,\ T_{this},\ \{d_{fj} \mapsto d_{fk}, \ldots\})$	By domain visibility
try $\mathsf{RO}(\{o\ldots\},\ d_{fk})$	R-Pull-Object
	R-New-Object or
$\exists$ $\mathsf{RO}(\{o\ldots\},\ d_{fk})$	R-Merge-Object

□

Lemma: Ownership Tree. Tree structure must follow order of object creation.

Proof. The owner of an object is set at creation time and has to be an existing domain on an existing object, so the ownership relation is well-founded and has no cycles. This assumes that `unique` is not part of the system (having `unique` and assignment of ownership after creation could create cycles).

Lemma: Binding Chains. Given a location ℓ such that $\Sigma[\ell] = C{<}\ell_o.n_1, \ldots{>}$, and ℓ_o such that $\Sigma[\ell_o] = T_o{<}\ldots{>}$. If the corresponding object creation expression is of the form `new` $C{<}\overline{d}{>}(\overline{v})$ with d_1 a formal domain, there exists a sequence of `new` $C_1{<}\overline{f_1}{>}(\overline{v_1})\ldots$`new` $C_n{<}\overline{f_n}{>}(\overline{v_n})$, with $CT(C_i) = C_i{<}\overline{\alpha}{>}$ and a chain of bindings $C{::}\alpha_1 \mapsto C_i{::}\alpha_i$, $C_i{::}\alpha_i \mapsto C_j{::}\alpha_j$, $\ldots$, $C_k{::}\alpha_k \mapsto T_o{::}n_1$.

Proof. By induction on the evaluation rules (using rules *R-New* and *T-New*) in FDJ.

Limitations. The proof assumes that objects are only created in locally visible domains or domain parameters: it does not reflect the existence of the `lent`, `shared` and `unique` domains [5]. Indeed, the OOG may not reflect an object marked `unique` until it is assigned to a specific domain. Thus, an inter-procedural flow analysis is needed to track an object from its creation (at which point it is `unique`) until its assignment to a specific domain. The current tool does not implement this flow analysis, so a `unique` object returned from a factory method must be annotated with the domain in which it should be displayed. Similarly, the flow analysis can determine what domain a `lent` object is really in. Again, objects annotated with `lent`, except for the root object, are currently missing from the OOG. Objects that are `shared` would be trivial to display in the OOG but would add many uninteresting edges (the analysis may also excessively merge objects in the `shared` domain), so we currently exclude them.

Assumptions. The OOG inherits other properties that are guaranteed by the soundness of the underlying ownership type system. For example, every object is assigned an owning domain which is consistent with all program annotations and does not change over time. These invariants are correct up to the following assumptions:

- **All Sources Available:** The program's whole source code is available, and the program operates by creating some main object and calling a method on it (this justifies the focus on a single root object, although multiple root objects could in principle be shown). The class of that main object is the type of the root of the OOG;
- **No Reflective Code:**Reflection and dynamic code loading may violate the above invariants by introducing unknown objects and edges, and possibly violating the guarantees of the underlying ownership system;
- **Flow Analysis:** Objects annotated with `shared` and `unique` are not currently shown in the OOG. Objects that are `shared` would be trivial to add but would add many uninteresting edges to the OOG. Objects that are `unique` would require a flow analysis to be handled properly. Usage edges (e.g., method invocations, field accesses) could be generated for a system with only ownership, but a flow analysis is required for usage edges to be sound in the presence of `lent` objects.

Despite the assumptions about the whole program source being available and restrictions on reflection and dynamic loading, our system is still relatively sound in the presence of these features. As long as the

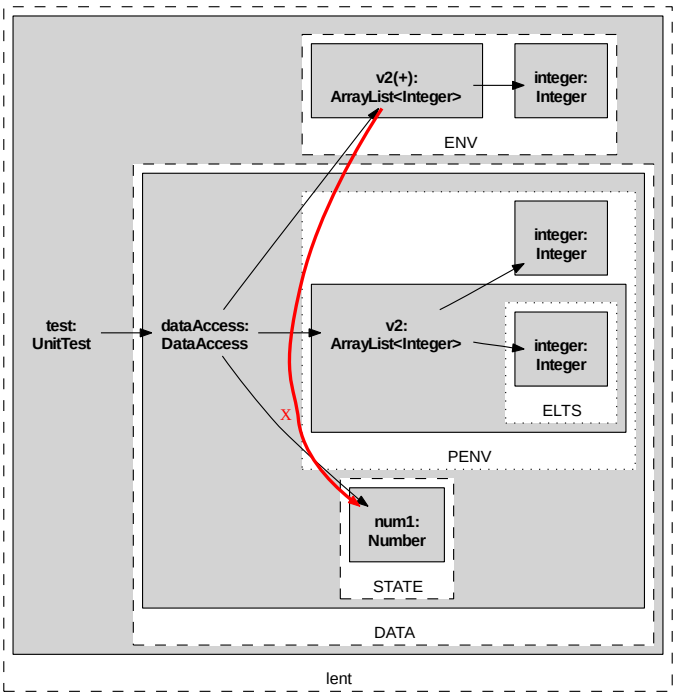


Figure 17: `DataAccess` OOG showing formal domains.

reflective operations are annotated correctly and consistently with ownership information, then any object referred to by some field in the source code that is available will show up in the OOG, as specified above.

4.3 Illustrative Example

Figure 18 shows a few representative rewrites that the reader can follow along with the OOG in Figure 17, which shows formal domains to clarify the binding of formals to actuals and the object pulling operation. We simplified the `ArrayList` inheritance hierarchy, by collapsing the abstract base class `AbstractList` and interface `List` into `ArrayList`, and eliding other types it implements, e.g., `Comparable` and `RandomAccess`.

We manually added to Figure 17 a thick edge labeled X as an example of an imprecise edge that the rules do *not* generate. Intuitively, `ELTS` is not bound to `STATE` so no rule should ever add an edge from `v2:ArrayList` to `num1:Number` in `STATE` — even though `Integer` is a subtype of `Number`. This makes OOG edges more precise than those obtained by superimposing field reference edges based on the associations in a class diagram. The rewriting rules here generate edges that are more precise than our earlier algorithm [2].

$(o_{num1}, d_{STATE}, t_{Number}, \{\})$ $(o_{int1}, d_{STATE}, t_{Integer}, \{\})$ — Field decl. `num1` and `int1` (1)

$(d_{STATE}, \{o_{num1}, o_{int1}\}, t_{DataAccess})$ — Domain decl. `STATE` and `PENV` (2)

$(d_{PENV}, \{o_{v2}\}, t_{DataAccess})$ — Domain decl. `STATE` and `PENV` (3)

$(o_{v2}, d_{PENV}, t_{ArrayList<Integer>}, \{d_{ELTS} \mapsto d_{PENV}\})$ — Field decl. `v2` (4)

$e_1 := (t_{DataAccess}, [d_{PENV}], t_{ArrayList<Integer>})$ — Field ref. `DataAcces` → `v2` (5)

$e_2 := (t_{DataAccess}, [d_{STATE}], t_{Integer})$ — Field ref. `DataAccess` → `int1` (6)

$e_3 := (t_{DataAccess}, [d_{STATE}], t_{Number})$ — Field ref. `DataAccess` → `num1` (7)

$(t_{DataAccess}, \{d_{STATE}, d_{PENV}\}, \{e_1, e_2, e_3\})$ — Type decl. `DataAccess` (8)

$(o_{dataAccess}, d_{DATA}, t_{DataAccess}, \{d_{PENV} \mapsto d_{ENV}\})$ — Field decl. `dataAccess` (9)

$(d_{DATA}, \{o_{dataAccess}\}, t_{UnitTest})$ $(d_{ENV}, \{\}, t_{UnitTest})$ — Domain decl. `DATA` and `ENV` (10)

$e_4 := (t_{UnitTest}, [d_{DATA}], t_{DataAccess})$ — Field ref. `UnitTest`→ `dataAccess` (11)

$(t_{UnitTest}, \{d_{DATA}, d_{ENV}\}, \{e_4\})$ — Type decl. `UnitTest` (12)

$(t_{Integer}, \{\}, \{\})$ $(t_{Number}, \{\}, \{\})$ — Type decl. `Integer` and `Number` (13)

$(o_{obj}, d_{ELTS}, t_{Integer}, \{\})$ — Field decl. for `ArrayList` element (14)

$(d_{ELTS}, \{o_{obj}\}, t_{ArrayList<Integer>})$ — Domain to store `ArrayList` elements (15)

$e_5 := (t_{ArrayList<Integer>}, [d_{ELTS}], t_{Integer})$ — Field ref. to `ArrayList` elements (16)

$(t_{ArrayList<Integer>}, \{d_{ELTS}\}, \{e_5\})$ — Type decl. for `ArrayList<Integer>` (17)

$(o_{test}, d_{lent}, t_{UnitTest}, \{\})$ — Variable decl. for `test` (18)

$RootObject := o_{test}$ — Root object (user selected) (19)

Sample rewrites to convert a few AbstractObjects into RuntimeObjects:

R-Convert-Object$(o_{test}) \hookrightarrow$ R-New-Object$(o_{test}) \hookrightarrow$ RuntimeObject$(\{o_{test}\}, d_{lent})$ (vo1)

R-Convert-Object$(o_{num1}) \hookrightarrow$ R-New-Object$(o_{num1}) \hookrightarrow$ RuntimeObject$(\{o_{num1}\}, d_{STATE})$ (vo2)

R-Convert-Object$(o_{int1}) \hookrightarrow$ RuntimeObject$(\{o_{num1}\}, d_{STATE}) \in \Theta$
and $o_{num1} : Number$ and $o_{int1} : Integer$ and $Integer <: Number$
$\hookrightarrow$ R-Merge-Objects$(o_{int1}) \hookrightarrow$ replace RuntimeObject$(\{o_{num1}\}, d_{STATE})$ with RuntimeObject$(\{o_{num1}, o_{int1}\}, d_{STATE})$ (vo3)

R-Convert-Object$(o_{dataAccess}) \hookrightarrow$ RuntimeObject$(\{o_{dataAccess}\}, d_{DATA})$ (vo4)

R-Pull-Object$((\{o_{v2}\}, d_{PENV}))$ and RuntimeObject$(\{o_{dataAccess}\}, d_{DATA}) \in \Theta$ and AbstractDomain$(d_{PENV}, t_{DataAccess})$
and aparam$(o_{dataAccess}, d_{PENV}, d_{ENV}) \hookrightarrow$ try$(\{o_{v2}\}, d_{ENV}) \hookrightarrow^*$ RuntimeObject$(\{o_{v2}\}, d_{ENV})$ (vo5)

Sample rewrites to convert a few AbstractEdges into RuntimeEdges:

$(\{o_{dataAccess}\}, d_{DATA})$ and AbstractEdge(5) (20)

$\hookrightarrow$ RuntimeEdge$([d_{DATA}], t_{DataAccess}, [d_{DATA}, d_{PENV}], t_{ArrayList<Integer>})$ (ve1)

Binding $d_{PENV} \mapsto d_{ENV}$ on (9) and R-Path-Subst maps $[d_{DATA}, d_{PENV}]$ to $[d_{ENV}]$ (newpath1)

$\hookrightarrow^*$ RuntimeEdge$([d_{DATA}], t_{DataAccess}, [d_{ENV}], t_{ArrayList<Integer>})$ (ve1.1)

$(\{o_{v2}\}, d_{PENV})$ and AbstractEdge on (16) (21)

$\hookrightarrow$ RuntimeEdge$([d_{PENV}], t_{ArrayList<Integer>}, [d_{PENV}, d_{ELTS}], Integer)$ (ve2)

Binding $d_{ELTS} \mapsto d_{PENV}$ on (4) and R-Path-Subst maps $[d_{PENV}, d_{ELTS}]$ to $[d_{PENV}]$ (newpath2)

$\hookrightarrow^*$ RuntimeEdge$([d_{PENV}], t_{ArrayList<Integer>}, [d_{PENV}], t_{Integer})$ (ve2.1)

Figure 18: Rewriting rules illustrated on the `DataAccess` example. The top-half of the diagram shows selected AbstractGraph tuples. The lower-half shows selected RuntimeObjects and RuntimeEdges that the rewriting rules create. $\hookrightarrow$ denotes the next generated fact, and $\hookrightarrow^*$ denotes the fact obtained at the fixed point.

1. For each type declaration C in the program
   (a) Create AbstractType t and add it to *Types*
   (b) For each actual or formal domain in C
      i. Create corresponding AbstractDomain d
      ii. Add d to $t.Domains$
   (c) Create default *owned* domain in t
   (d) For each domain link between d_1 and d_2 in C
      i. Create AbstractLink between the AbstractDomain of d_1 and the AbstractDomain of d_2
      ii. Add AbstractLink to $t.Links$
   (e) For each declaration $d\ C'\langle\overline{a}\rangle\ \ o$ in C **(DBV) or else**
   (f) For each creation $new\ C'\langle\overline{a}\rangle(\ldots)$ in C **(IBV)**
      i. If C' has no AbstractType, create t' for C'
      ii. If AbstractType t of Type C has no AbstractDomain d, create d and add d to $t.Domains$
      iii. Create AbstractObject o and add to $d.Objects$
      iv. Create bindings $\{b\ldots\}$ from formals $\overline{f}$ of AbstractType t' to actuals $\overline{a}$ of t and add to $o.Bindings$
      v. If field declaration (in DBV) or object creation assigned to a field (in IBV)
         A. Create AbstractEdge e from AbstractType t to AbstractDomain a_1 and AbstractType t'
         B. Add e to $t.Edges$
2. Unify domains related in an inheritance hierarchy
   (a) If $C <: T$, unify domains $C::d$ and $T::d$
   (b) If interface I declares public domain $I::d$, unify with $C::d$ if C implements I
3. Expand generic types (perform type substitutions)
4. Synthesize AbstractEdges from array type to array element

Figure 19: Visitor to generate the AbstractGraph.

5 Abstraction by Types

We motivate several advanced features of the analysis using a real system, JHotDraw, that we revisit in the evaluation. Section 6.1 highlights how we annotated JHotDraw. We now discuss additional features of the analysis to provide abstraction by types, in addition to abstraction by ownership hierarchy. We motivate these features using a real object-oriented system.

We motivate and extricate abstraction by types from the core algorithm. The rewriting rules already include it (See Rule R-Aux-Compat). The instantiation-based view mainly requires constructing the AbstractGraph differently (See Figure 19) but the transformation from the AbstractGraph to the RuntimeGraph stays mostly the same.

JHotDraw is rich with design patterns, uses composition and inheritance and has evolved through several versions. Version 5.3 has 200 classes and 15,000 lines of Java. We defined three top-level domains to organize the core types as follows:

- `MODEL:` has instances of `Drawing`, `Figure`, `Handle`, etc. A `Drawing` is composed of `Figures`. A `Figure` has `Handles` for user interactions;
- `VIEW:` has instances of `DrawingEditor`, `DrawingView`, etc.;
- `CONTROLLER:` has instances of `Tool`, `Command` and `Undoable`. A `DrawingView` uses a `Tool` to manipulate a `Drawing`. A `Command` represents an action to be executed.

5.1 Instantiation-Based View

JHotDraw uses inheritance heavily, whereby many types 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`.

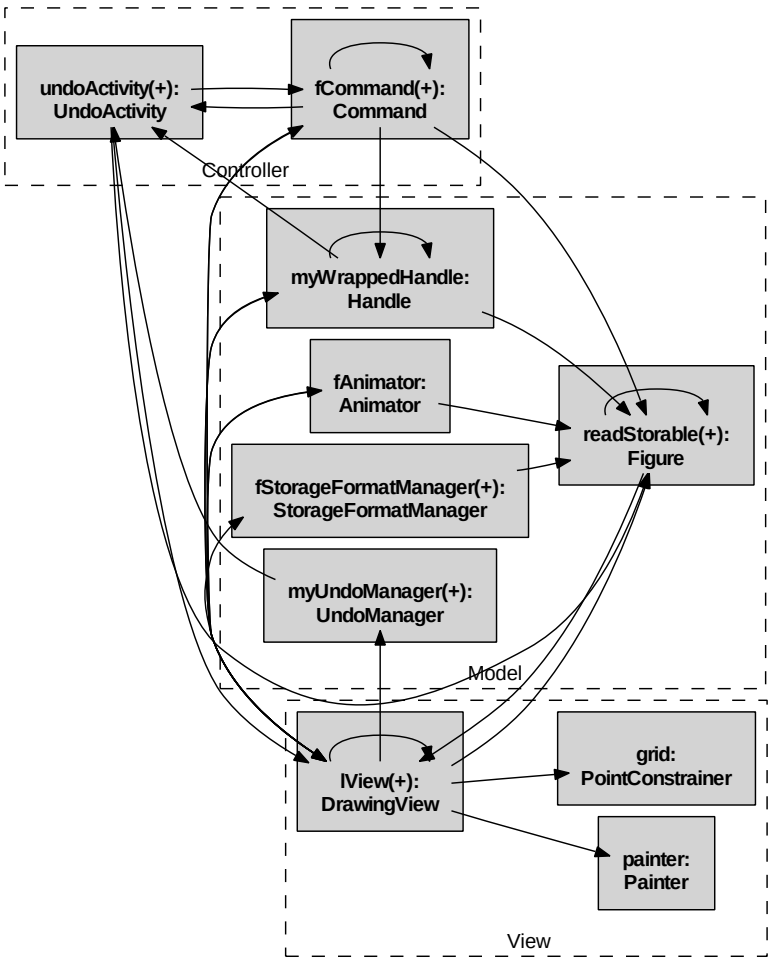


Figure 20: JHotDraw OOG with a declaration-based view.

Consider $\theta_{Tool} = (\{o_{Tool}, o_{VCL}, \ldots\}, d_C)$ and $\theta_{Cmd} = (\{o_{Cmd}, o_{VCL}, \ldots\}, d_C)$ with o_{Cmd}:`Command`, o_{Tool}:`Tool` and o_{VCL}:`VCL`. `Command` <: `VCL` and `Tool` <: `VCL` but neither `Tool` <: `Command` nor `Command` <: `Tool`. `VCL` is `ViewChangeListener` and d_C is `CONTROLLER`. R-MERGE-EXISTING replaces θ_{Tool} and θ_{Cmd} with $\theta_{ToolCmd} = (\{o_{Cmd}, o_{Tool}, o_{VCL}, \ldots\}, d_C)$.

As a result, the analysis merges the abstract objects for `Command` and `Tool` into the same runtime object. In keeping with the good practice of programming to an interface instead of an implementation, many abstract objects have interface types. As a result, the OOG that the analysis produces for JHotDraw merges too many architecturally relevant objects (See Figure. 20).

A key insight, however, is that there are no object creations of interface types. To regain some precision, we construct the AbstractGraph differently (Fig. 19). Line (c) generates a declaration-based view (DBV) by generating abstract objects for all field and variable declarations. In contrast, Line (d) generates an instantiation-based view (IBV) by considering only object creation expressions. This is similar to how Rapid Type Analysis (RTA) determines a method call's receiver during call graph construction [8].

In the example above, the analysis never generates an abstract object of type `ViewChangeListener`. Rather, it creates abstract objects for types `SelectionTool` and `AlignCommand`. When constructing the runtime graph, `AlignCommand` and `SelectionTool` are kept distinct since there is no subtyping relation between them. This achieves the goal of keeping `Command` and `Tool` distinct (`SelectionTool` <: `Tool`, `ViewChangeListener` and `AlignCommand` <: `Command`, `ViewChangeListener`).

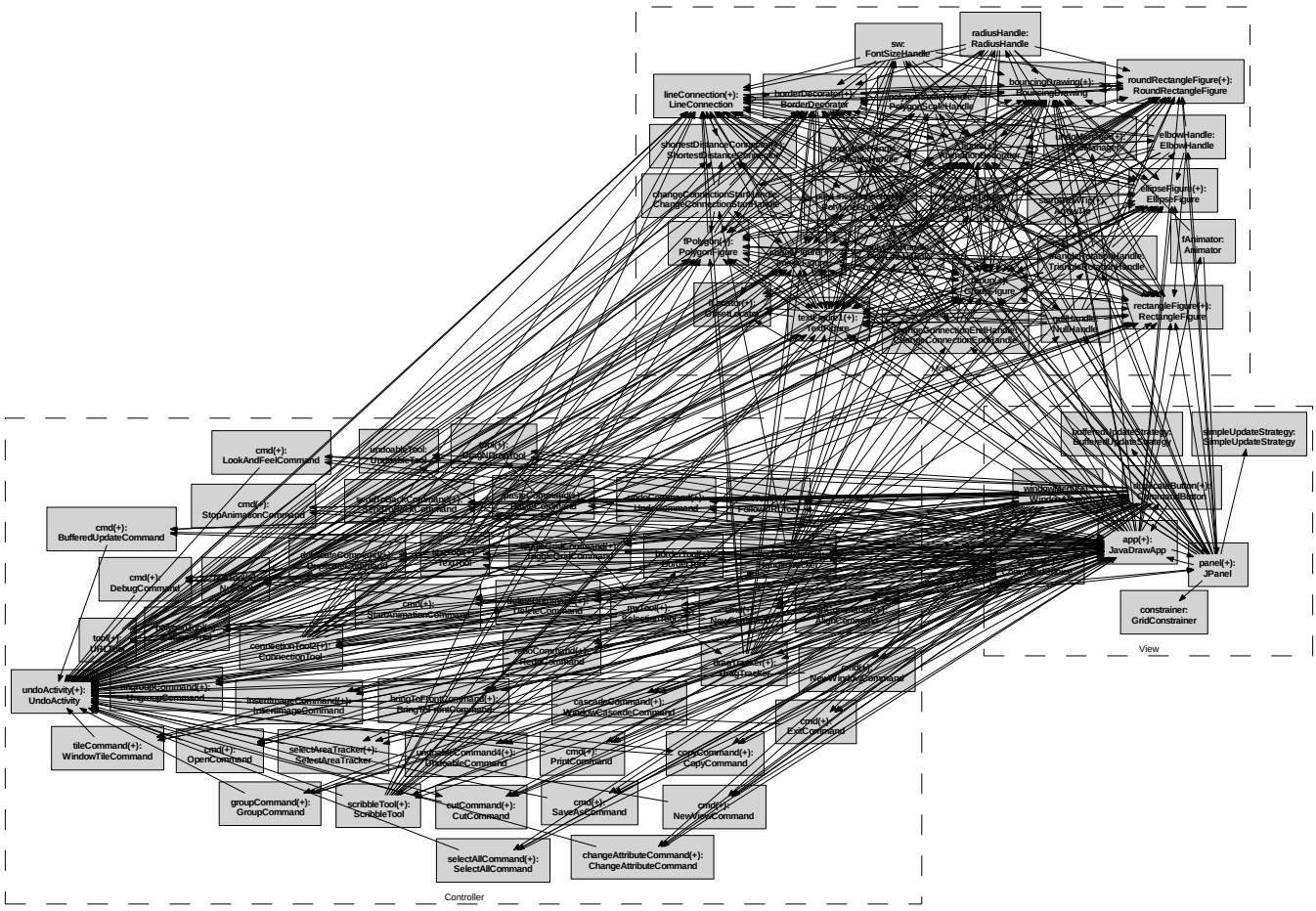


Figure 21: Thumbnail of the JHotDraw OOG based on object instantiations (but without abstraction by types). The embedded image becomes readable after zooming in by 800%.

An instantiation-based view as discussed above lacks abstraction because it shows objects for `RedoCommand`, `NewViewCommand`, etc., as well as objects for `ConnectionTool`, `CreationTool`, etc (See Figure 21). 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 soundness, the analysis adds an edge from or to each object in the source or target path that is type compatible with the source or target type. So the analysis adds an edge from `CommandMenu` to `RedoCommand`, `NewViewCommand`, etc. Moreover, a `Command` wraps another `Command`. So this results in an almost fully connected graph. Needless to say, this OOG is hardly an improvement over the earlier object graph (Figure 2) that Womble obtains from bytecode without any annotations!

Special Cases. Even in an instantiation-based view, the analysis must still handle variable declarations of interface types. For example, in JHotDraw, `CommandMenu` declares a `Vector<Command>`. Due to the presence of a virtual field, the analysis must still create a `Command` abstract object inside `Vector`'s `ELTS` formal domain that stores the elements. Recall that `Command` is an interface. The analysis cannot pick a more precise concrete type. The analysis also cannot simply ignore these abstract objects, as it must pull them, and have them carry the binding information to generate the appropriate RuntimeEdges. For instance, `ELTS` is transitively bound to `CONTROLLER`; after pulling a `Command` abstract object from `ELTS` to `CONTROLLER`, the analysis creates a RuntimeEdge from a `CommandMenu` object inside `VIEW` to any subclass of `Command` inside `CONTROLLER`, such as `RedoCommand`. Simply adding a `Command` abstract object to the `ELTS` domain would result in excessive merging as in a declaration-based view. Instead, the analysis creates a *virtual* abstract

object, one that gets pulled just like any other. But the analysis excludes a virtual abstract object from the list of objects inside an AbstractDomain — except to avoid re-adding it to that same domain. However, a virtual abstract object does not affect object merging. Finally, when creating the depth-limited projection of the runtime graph, the analysis omits virtual abstract objects after they have served their purpose. For simplicity, we exclude virtual objects from the formal system.

Finally, even when using an instantiation-based view, an object creation expression of the form `new Object()` would create an abstract object that would cause the analysis to merge all the objects in that domain into one object. To avoid this problem, the abstract graph construction synthesizes for such an abstract object an abstract type similar to an implicit anonymous class.

5.2 Abstraction by Trivial Types

To improve abstraction and reduce clutter, one heuristic makes the analysis merge abstract objects whenever they share one or more non-trivial *least upper bound (LUB) types*. The resulting runtime object has an intersection type that includes all the least upper bounds. This heuristic can be turned off by taking out the existsNonTrivialLUB disjunct in R-Aux-Compat (Fig. 14).

Merging all the abstract objects in a domain into a single runtime object of type `Object` would result in a sound but uninteresting OOG. So the heuristic does not merge abstract objects that only share *trivial* types as supertypes. Trivial types are user-configurable and typically include `Object`, `Cloneable` and `Serializable` from the Java Standard Library. Many marker interfaces that do not declare any methods, such as `RandomAccess`, are good candidates.

The result of using abstraction with default trivial types on JHotDraw is in Figure 22, and again, suffers from the same excessive merging as when not using the instantiation-based view.

We can achieve better results for JHotDraw by carefully selecting the trivial types. JHotDraw has its own list of interfaces that many classes implement such as `Storable` and `Animatable`. We added those to the list of trivial types, as well as constant interfaces such as `SwingConstants` (inheriting from a constant interface is a bad coding practice that predates Java 1.5 static imports). We also added the listener interfaces as trivial types. Based on the discussion above, the analysis merges `RedoCommand` and `NewViewCommand`, because `Command` is their non-trivial LUB, and similarly, for `ConnectionTool` and `CreationTool`. But the analysis does not merge `ConnectionTool` and `RedoCommand` because their LUB, `ViewChangeListener`, is a trivial type. The result of using abstraction with the non-default trivial types on JHotDraw is in Figure 23.

5.3 Abstraction by Design Intent 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. 27). 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 abstract objects 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.

To achieve this precision, the analysis also supports the following heuristic. A developer defines a list of design intent types. To decide whether to merge two abstract objects $o : t$ and $o' : t'$, the analysis finds the first design intent types in the list, $\hat{t}$ and $\hat{t'}$, such that $t <: \hat{t}$ and $t' <: \hat{t'}$. The analysis merges objects o and o' if $\hat{t'} <: \hat{t}$ or $\hat{t} <: \hat{t'}$. If the design intent type list does not include a type for t or t', then this heuristic does not apply. This heuristic corresponds to the disjunct mapToSameDIT in R-Aux-Compat and can also be turned-off.

JHotDraw's `framework` package includes abstract classes and interfaces that define the core framework. We added to the list of design intent types all the types in the `framework` package and ordered them from

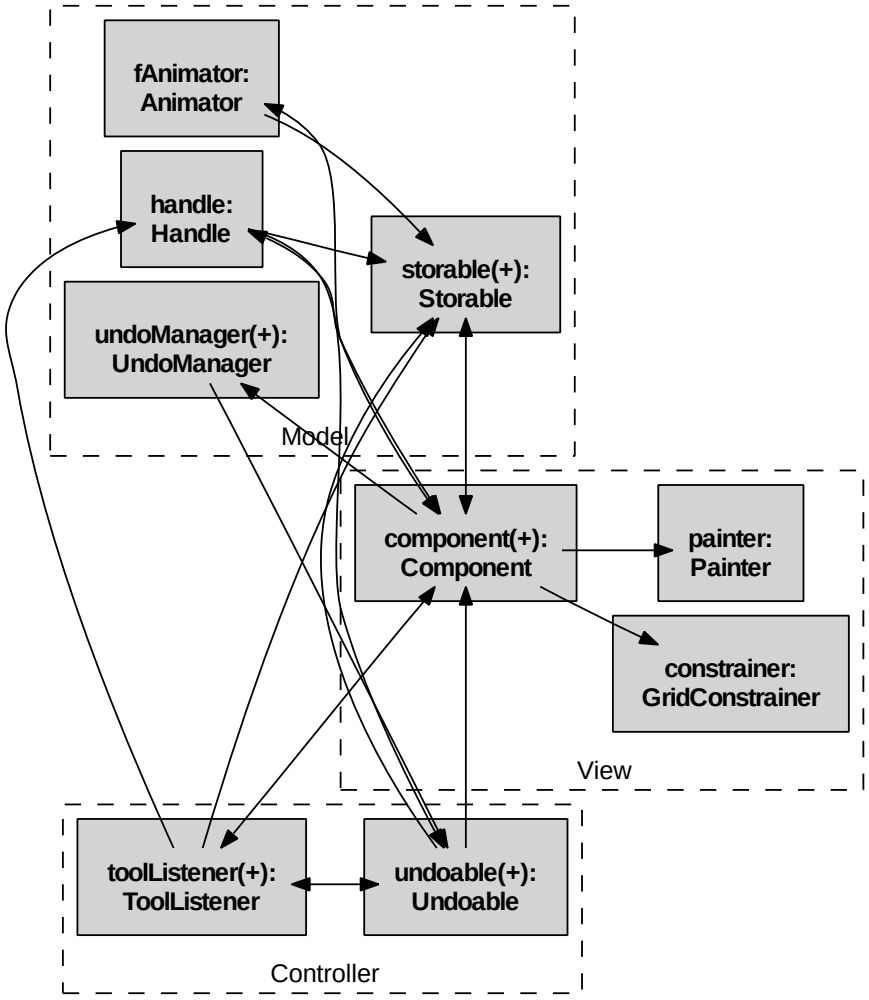


Figure 22: JHotDraw OOG with abstraction by trivial types (the default list).

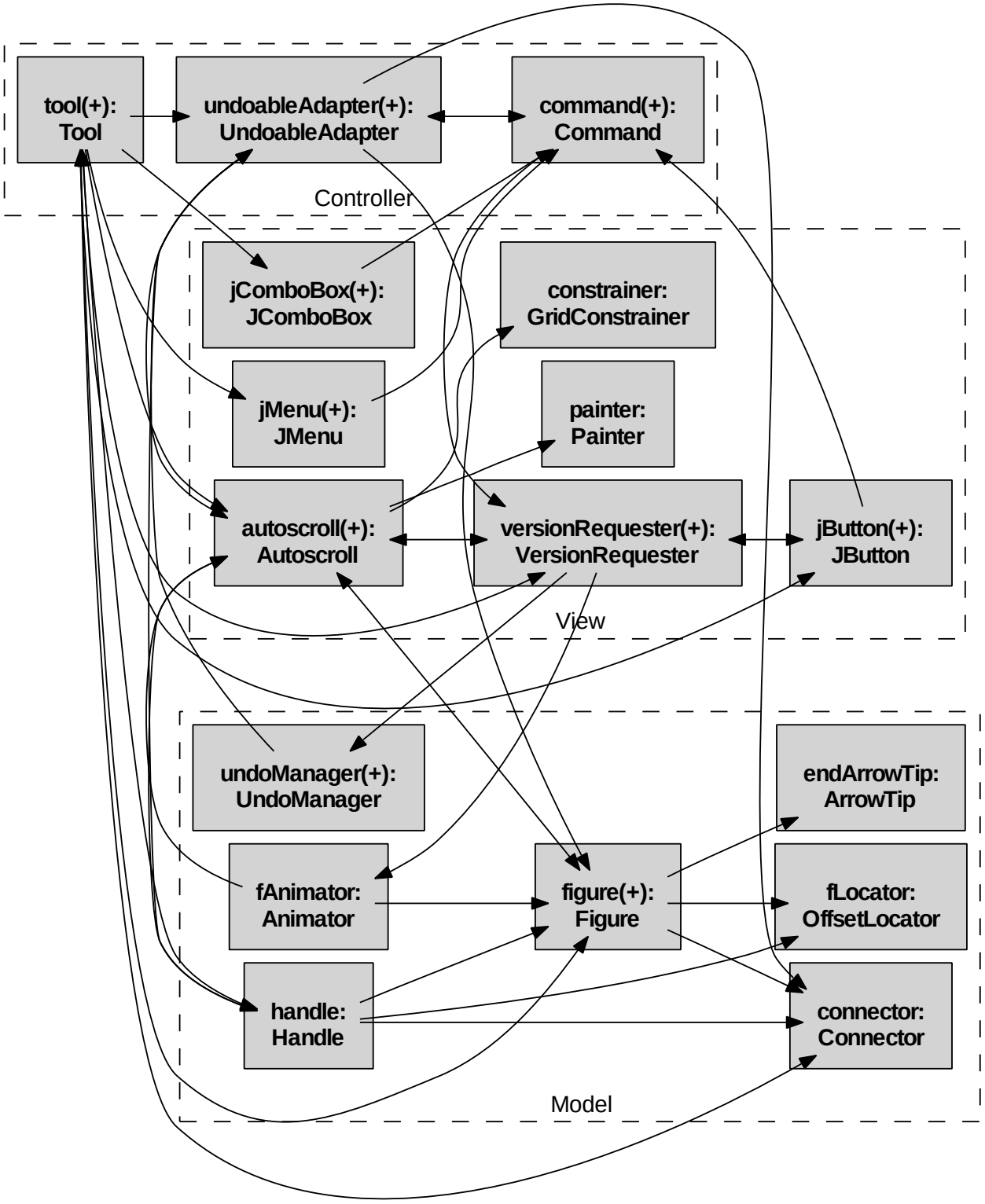


Figure 23: JHotDraw OOG with abstraction by trivial types (the fine-tuned list).

most to least architecturally relevant (`Drawing` appears before `Figure`). Abstraction by types merges objects of type `StandardDrawing` and `BouncingDrawing` with objects of type `Drawing` into one object. This heuristic also merges objects of type `AbstractFigure`, `CompositeFigure`, among others, with objects of type `Figure`. But it keeps objects of type `Drawing` and `Figure` distinct in `MODEL`, just as we desired.

Finally, it is worth noting that since abstraction by types leads only to additional merging of objects, it does not compromise soundness (unsoundness would mean showing two components for the same runtime object).

6 Evaluation

We evaluated the quality of the extracted OOGs on several extended examples of medium-sized representative programs to answer the following research questions:

- Can an OOG have a meaningful level of abstraction (or does it suffer from too much or too little merging)?
- Based on an OOG, can a developer learn what annotations she can adjust to get a desired architectural view?
- Can an automatically extracted OOG be comparable to a runtime architecture manually drawn by a developer?

Methodology. The tool support for architectural extraction consists of two Eclipse plugins. JavaDomains is a typechecker to validate ownership domain annotations that a developer inserts as Java 1.5 annotations [3]. OOG Wizard is a plugin to extract an OOG. It allows the developer to select the projection depth, elide substructure on selected objects and set the trivial types (see screenshot in Figure 25). The OOG is laid out automatically using GraphViz [19].

The study's subject (one of us, hereafter "we") developed the OOG Wizard but none of the subject systems. He mostly learned their architectural structure from iteratively annotating the code, examining the extracted OOGs and relating the OOGs to diagrams drawn by others. For one system (JHotDraw), he had access to a tutorial by the original designers, but for a slightly older version than the one he annotated. The tutorial discusses the design patterns that JHotDraw implements but does describe the system's runtime architecture. We previously studied another subject system (HillClimber) by re-engineering it to ArchJava [4]. The re-engineering case study also produced a version that cleaned up the original code, for instance by making most class fields as private. For this case study, we started from the refactored Java version and added ownership domain annotations to it. We discuss in detail the annotation process of the subject systems elsewhere [3]. In particular, for the HillClimber subject system [3], we discuss the differences between adding ownership annotations to the plain Java program, compared to the re-engineered ArchJava program.

6.1 JHotDraw

JHotDraw is rich with design patterns, uses composition and inheritance and has evolved through several versions. Version 5.3 has 200 classes and 15,000 lines of Java.

Design documentation for JHotDraw is available, e.g,. [17, 40, 22]. The class diagram in Figure 26 shows some of the core types. An often cited article [22] discusses how JHotDraw follows the Model-View-Controller design pattern (the package structure does not reveal that fact since all the types in Figure 26 are in one `framework` package).

Annotations. We 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 `Figure`s. A `Figure` has `Handle`s for user interactions;
- `VIEW: DrawingEditor`, `DrawingView`, etc., instances;
- `CONTROLLER:` has instances of `Tool`, `Command` and `Undoable`. A `DrawingView` uses a `Tool` to manipulate a `Drawing`. A `Command` represents an action to be executed — the Command pattern without undo.

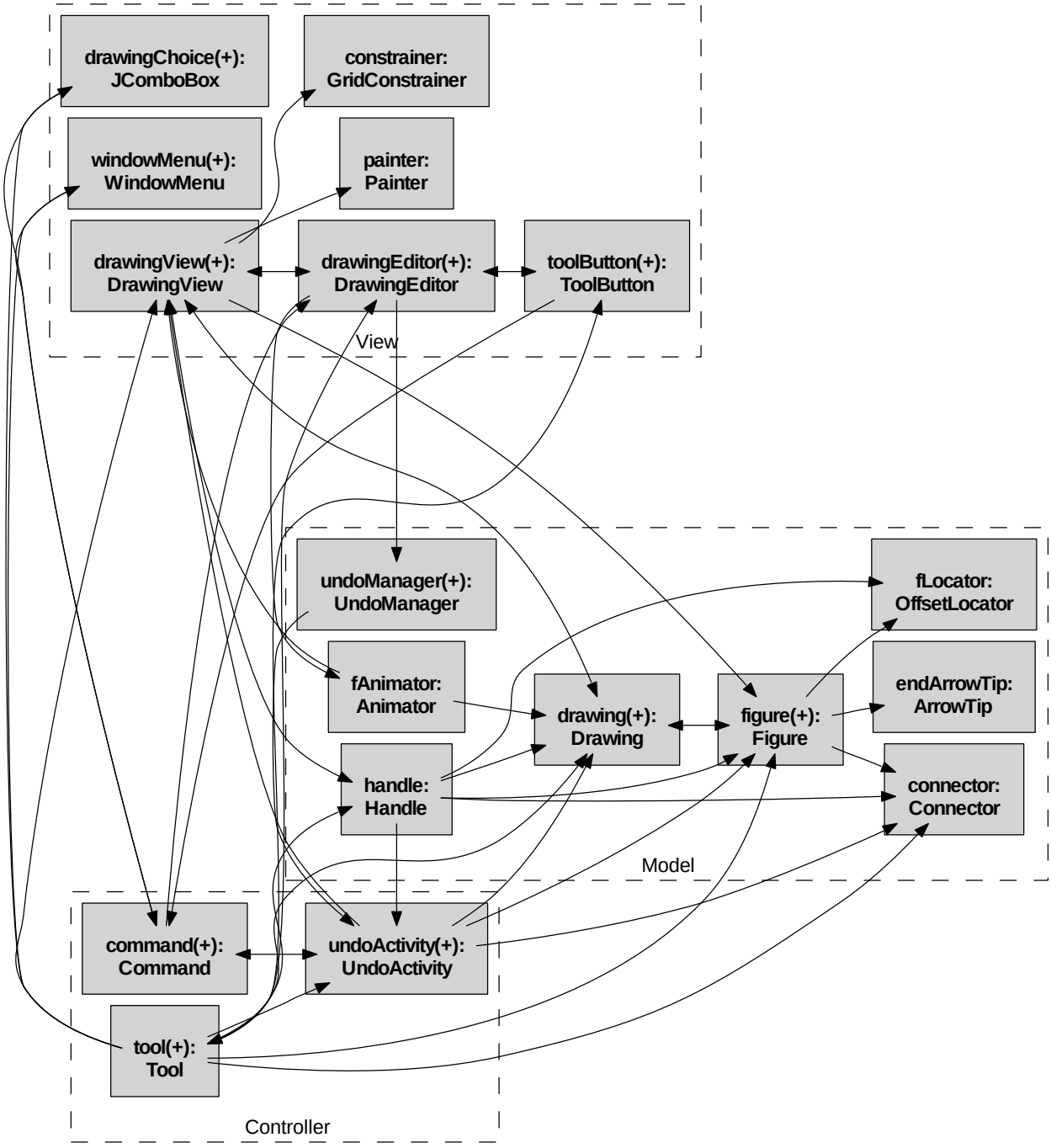


Figure 24: JHotDraw OOG with abstraction by design intent types.

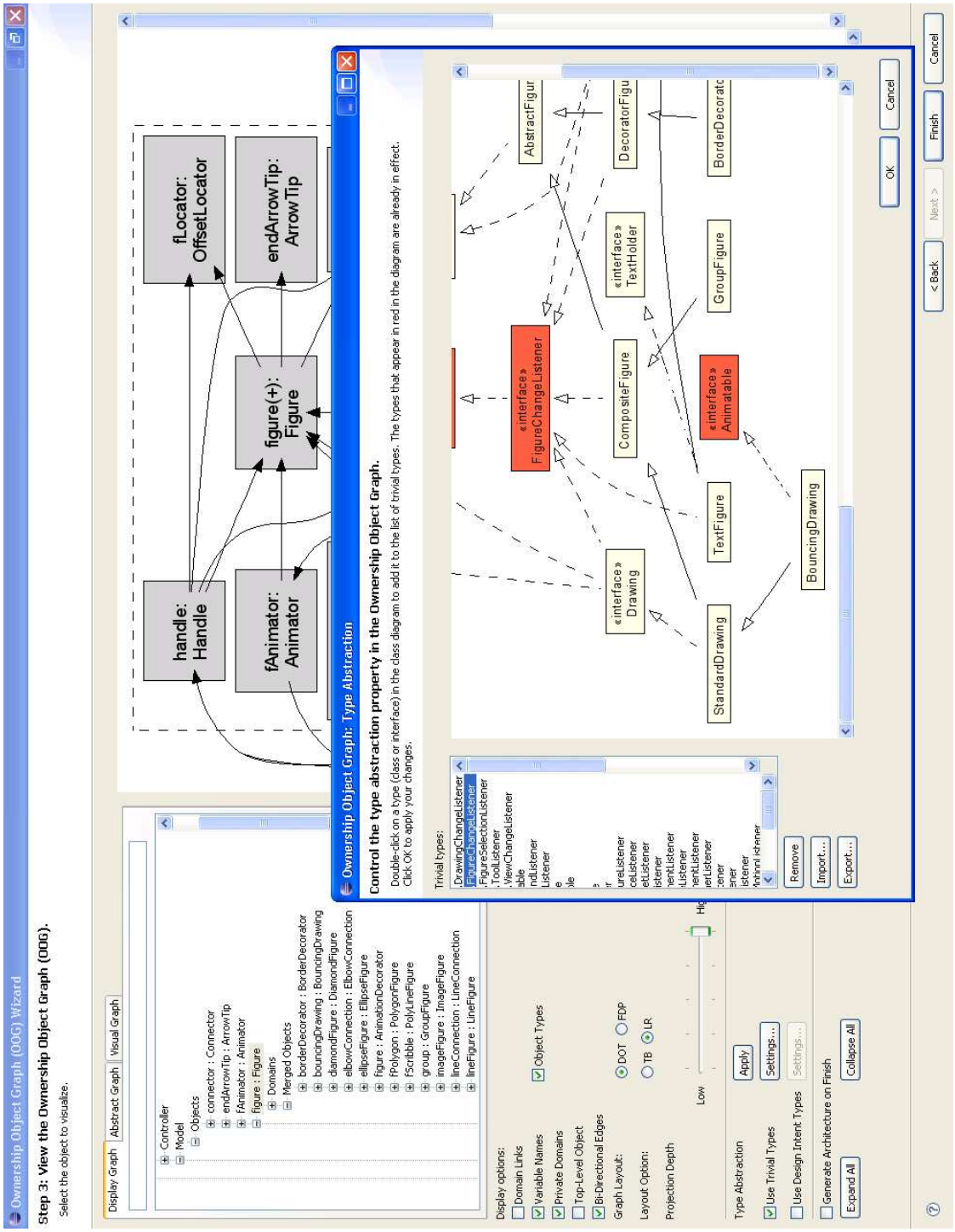


Figure 25: The left pane shows the ownership tree and the right pane shows the depth-limited projection. The tool shows the abstract objects merged into a runtime object and helps a developer set the trivial types for the abstraction by types.

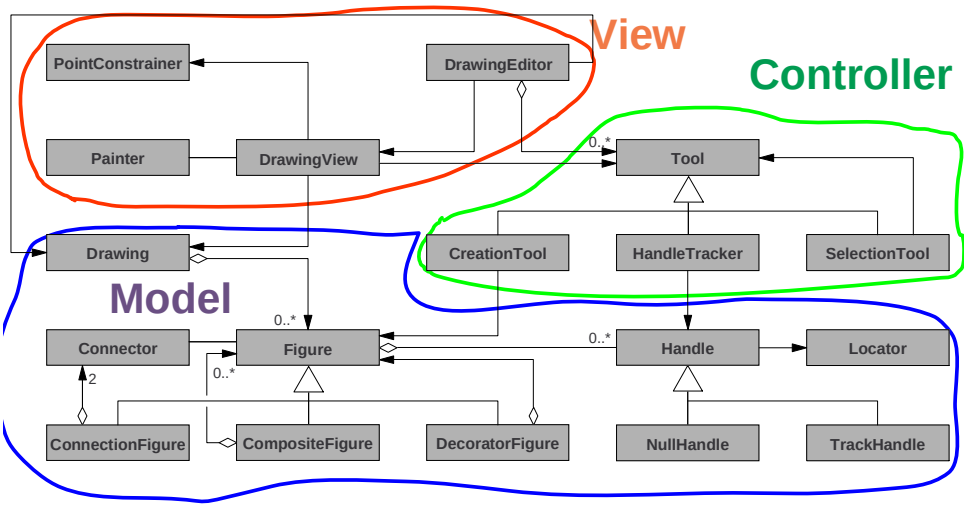


Figure 26: JHotDraw class diagram showing how we annotated instances of the selected types (Source: [40]).

Evaluation. In Section 5, we showed some earlier OOGs we extracted for JHotDraw. The OOG is in Figure 20 suffered from too much merging: it merged `DrawingView` and `DrawingEditor` and made it hard to recognize many of the key abstractions from the class diagram in Figure 26. Moreover, the OOG in Figure 21 was unreadable and required abstraction by types.

Abstraction by Types. We turned on abstraction by trivial types, initially using the default list of trivial types. This results in an OOG where each visual object merges too many abstract objects. The developer selects the trivial types as follows. First, he graphically selects an object which appears to merge too many objects. The OOG Wizard then displays an inheritance hierarchy of the types of the abstract objects that are merged into the selected object. 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. This results in fewer objects getting merged into an abstract object.

For JHotDraw, we started with the list of default trivial types, which includes several interfaces from the Java Standard Library such as `Serializable`, `Cloneable`, etc. Many are marker interfaces that do not declare any methods. Others are constant interfaces that only define constants, such as `SwingConstants`[9]. JHotDraw had its own list of interfaces that many classes implement such as `Storable` and `Animatable`. We also included several listener interfaces, such as `ViewChangeListener` (as discussed in Section 5.1). Because of JHotDraw's complex inheritance hierarchy, the list of trivial types needed fine-tuning to achieve the desired level of abstraction — more so than the other subject systems. We did not use abstraction by types on the last subject system (Section 6.3).

Riehle previously studied JHotDraw and produced the code architecture in Figure 26. 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* — although important for code reuse, such a code factoring is often unimportant from a design standpoint; and (b) *subsume a set of similar classes under a smaller set of representative classes* — showing many similar subclasses that vary only in minor aspects often leads to needless clutter [40, 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`, or any of its concrete subclasses `ElbowHandle`, `NullHandle`, etc., appear as one `Handle` component 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 Figure 26 indicates. Only `NullHandle` and its subclasses do. But since they were all merged

[9]Inheriting from a constant interface is a bad coding practice, the Constant Interface *antipattern* [9, Item #17], and Java 1.5 supports *static imports* to avoid it. This is one more reason to avoid it!

into `Handle`, the OOG shows an edge from `Handle` to `Locator` in Figure 27.

We were slightly surprised when we 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 sound: there is no runtime object that can have both types `NullHandle` and `AbstractHandle`, so one runtime object does not appear as two in the OOG.

Potential Design Flaw. We were surprised that the OOG did not show distinct `Drawing` and `Figure` objects, presumably core types in the class diagram in Figure 26.

We used the tool to determine that one object in the `Model` domain merged both `Drawing` and `Figure`. We examined the type hierarchy and learned that the base class implementing the `Drawing` interface, `StandardDrawing` extends `CompositeFigure`. Thus a `Drawing` *is-a* `Figure`. We researched this finding and found a brief mention in the Version 5.1 Release Notes. Still, in the `framework` package, interface `Drawing` does not extend `Figure`! We then checked the JHotDraw tutorial. Indeed, the JHotDraw designers explicitly asked to "not commit to the `CompositeFigure` implementation since some applications need a more complicated representation" [17, Slide #16].

OOG = Architecture? The OOG in Figure 27 seems to have the right level of abstraction since we recognize in it most of the core types from Figure 26.

There are three top-level domains: Model, View and Controller. Object `Figure` merges objects of type `Figure`, `TriangleFigure`, etc. Because a `Drawing` is implemented as a `Figure`, object `Figure` also merges objects of type `Drawing`, `StandardDrawing`, etc. The `DrawingView` interface extends the `DrawingChangeListener` interface. Hence the edge from object `fListeners` inside object `Figure` to the `DrawingView` object. Inside object `Figure`, object `fFigures` contains the composite `Figure` objects. Object `Handle` merges objects of type `NullHandle`, `GroupHandle`, etc. `Point` objects are immutable and passed linearly, hence they do not appear in the OOG.

The number of components in each tier is typical of architectural diagrams [24]: `Model` has 7 components; `View` has 7 components; and `Controller` has 3 components. More precise annotations could split the `Model` domain into a domain for *application model* objects, such as `UndoManager` and `StorageFormatManager`, and one for *domain model* objects such as `Figure` and `Handle` objects, as in the Model-Model-View-Controller pattern[10].

A key issue in architectural extraction is distinguishing between architecturally relevant and non-architecturally relevant objects. The OOG provides architectural abstraction by folding lower-level objects into higher-level architectural components. As a result, the OOG does not show non-architecturally relevant objects in the top-level domains. Collapsing many nodes into one is a classic approach to shrink a graph. However, the OOG statically collapses nodes based on the execution and ownership structure, and not according to where objects were declared in the program, or according to some naming convention. Abstraction by types causes more collapsing, but it applies only within a domain, i.e., it never merges two objects in two different domains.

There are two ways to control the level of detail. One is to control the OOG projection depth, which affects the depth of object substructure uniformly for all objects starting from the root of the ownership tree (slider control in the OOG Wizard in Figure 25). Because one object's substructure may be more interesting than that of some other object, the OOG Wizard tool 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 Figure 27, we manually elided the substructure of all the objects in the top-level domains except for `Drawing` because we wanted 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 summary edge from `fFigures` to `figure:Figure` in `Model`.

In principle, one could manually elide objects in WOMBLE's output (Figure 2) to obtain a more abstracted diagram. One could also apply heuristics similar to the ones we used for abstraction by types. Indeed, in many architectural recovery approaches, the developer filters elements that satisfy certain query criteria to produce more abstracted views — e.g., collapse all nodes labeled with a common prefix according to some

[10] `http://c2.com/cgi/wiki?ModelModelViewController`

naming convention into a single subsystem [47]. 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 not clear how the developer can decide which implementation details can be elided. Indeed, O'Callahan mentions that object graphs tend to expose the implementation of data structures [33, p. 252].

6.2 HillClimber

By many accounts, JHotDraw is the brainchild of object-oriented analysis and design (OOAD) experts. The second subject system HillClimber is a 15,000 line Java application that was developed by undergraduates. HillClimber is also interesting because it uses a framework and its architectural structure had degraded over the years [3]. Our goal was to evaluate the OOG of a program that was not well-designed by OOAD experts. In HillClimber, the application *window* uses a *canvas* to display *nodes* and *edges* of a *graph* to show the output of a computational *engine*.

Annotation. The ownership annotations organized objects into a `data` domain to store the `graph`, a `ui` domain to hold user interface objects, and a `logic` domain to hold the `engine`, search objects, and associated objects. While adding annotations to HillClimber, we refactored the code to reduce coupling between `ui` and `data` objects [3].

Evaluation. The reader can compare the HillClimber OOG in Figure 28 to the one we previously published [2]. The tool shows the abstract objects merged into a runtime object (Figure 25). We used that information to learn what abstract objects in the program required different annotations. We refined those annotations using the following two strategies.

Strategy #1: Use encapsulation. We reduced the clutter in the `dataTier` 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 `Vector`s 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.

Strategy #2: Use logical containment. We defined public domains to reduce the number of top-level objects. A public domain groups related objects, pushes the inner objects it contains down the ownership tree and removes them from the top-level domains, while keeping those inner objects accessible to objects that can access the outer object. Object `search` has a `HEURISTICS` public domain with two array objects inside it; its peer object `heuristics` inside `logicTier` accesses those array objects directly[11].

6.3 Aphyds

Aphyds is an 8,000 line circuit layout application that Aldrich et al. studied previously [6]. Aphyds follows the Document-View style where the views are user interface objects, and the model consists of a *circuit* and computational objects to partition and route the circuit.

Annotations – Round 1. We initially organized the Aphyds objects into two top-level domains:

- `UI`: containing a `CircuitViewer` object and several subsidiary user interface objects;
- `Model`: holds a `Circuit` object and a set of computational objects that act on it, such as `Floorplanner` and `Partitioner`.

Evaluation – Round 1. These annotations produced an OOG with too many components in the top-level domains (Figure 6.3). This OOG is not very comparable to the diagram drawn by the original Aphyds developer (Figure 1). In particular, objects `Circuit`, `Net`, `Terminal` and `Node` are at the same level.

Annotations – Round 2. We examined the OOG and determined which runtime objects needed to be pushed down the hierarchy. Using the tool, we learned what abstract objects declared in the program required different annotations. For instance, we needed to push `Net` and `Node` objects underneath `Circuit`. Using Figure 1 as a guide, we created public domains as follows:

[11] An owner-as-dominator type system [12] would not allow such an edge.

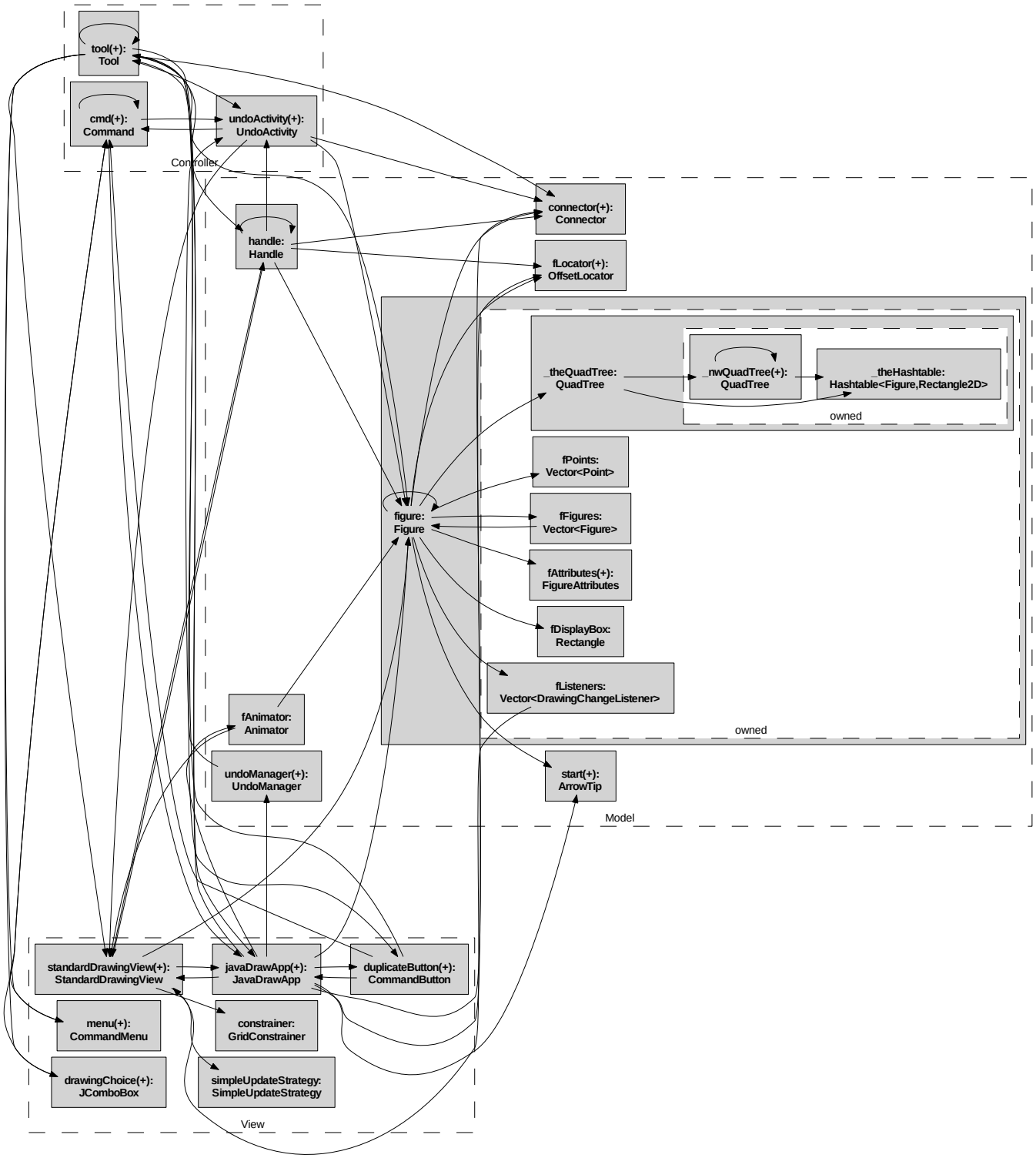


Figure 27: Top-level JHotDraw OOG. The objects in the top-level domains are collapsed, except for `Figure`.

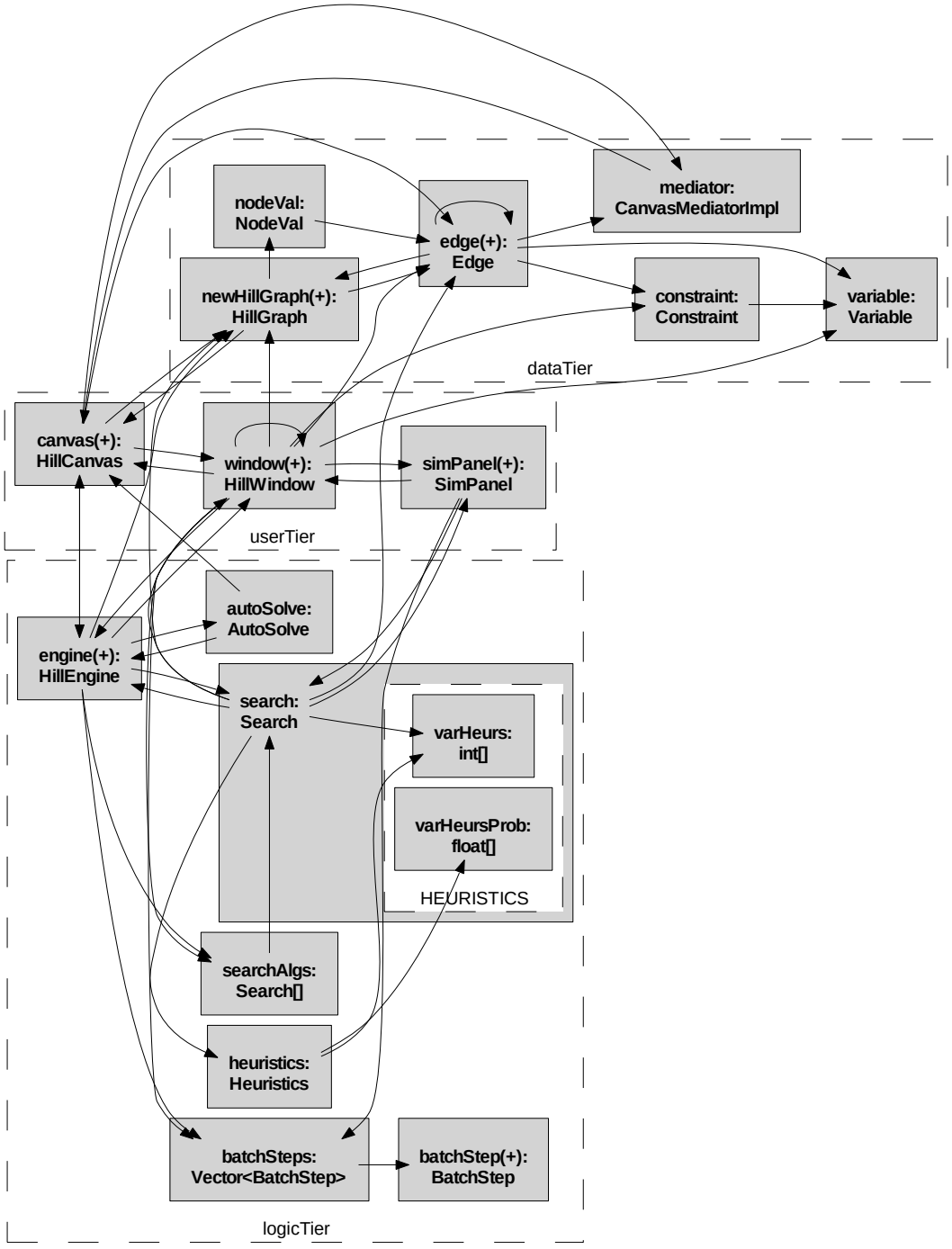


Figure 28: Top-level HillClimber OOG. The objects in the top-level domains are collapsed, except for `search`.

- `CircuitViewer.DISPLAY`: a public domain on the `CircuitViewer` to hold a `Displayer` object that all the other UI objects had references to;
- `Circuit.DATABASE`: a public domain on the `Circuit` object, that includes the objects associated with the `Circuit`, such as `Net`, `Node`, and `Terminal`;
- `Partitioner.DATABASE`: a public domain on `Partitioner` to hold `PartitionTranscript` and `PTnode` objects;
- `Floorplanner.DATABASE`: a public domain on `Floorplanner` for the floorplanning objects, such as `SlicingTree`;
- `GlobalRouter.DATABASE`: a public domain on `GlobalRouter` to hold `NetGlobalRouting` objects.

Just as with HillClimber, we also reduced the clutter by pushing objects such as `Vector<Floorplan>` into private domains or by passing them linearly between objects.

Evaluation – Round 2. The Aphyds OOG with the revised annotations is in Figure 6.3. There are two top-level domains, `UI` and `Model`. Many objects that were in the `Model` domain were moved into public domains of other objects in the `Model` domain, such as `Channel`, `GlobalRouter`, `Partitioner` and `Circuit`. Those public domains are elided except for the `Circuit` object. Inside object `Circuit`, public domain `DATABASE` has `Node`, `Net` and `Terminal` objects inside it. The `owned` domain inside `Circuit` stores `Hashtable` objects.

Indeed, this OOG is very comparable to the developer diagram of the Aphyds runtime architecture (Figure 1). For instance, `viewer`, `circuit` and `fp` in the OOG map to `circuitViewer`, `Circuit` and `FloorPlanner` in Figure 1. Objects `Node`, `Net` in `Circuit`'s public domain `DATABASE` map to `node` and `net` inside `circuit`'s sub-architecture in Figure 1.

The up-to-date extracted OOG somewhat contradicts the documented runtime architecture. As to be expected from a manually generated diagram, Figure 1 omitted several edges between `UI` and `Model` objects as well as edges between objects in the `Model` tier. Many connections thought to be unidirectional in the developer's diagram, such as between the `CircuitViewer` and the various `Dialog`s, turned out to be bi-directional in the implementation. The OOG has an additional object in `UI`, `PartDialog`, which connects to the `Partitioner`. Upon a closer examination of the OOG, we noticed a reference from `placer` in `Model` to `PlacerDialog` in `UI`. This was a potential red flag since Aphyds is a multi-threaded application: a worker thread executing long running operations cannot carelessly call back into the user interface thread. The traceability information in the OOG helped us relate this callback to a field of type `PlaceRouteDialog` declared in class `Placer`. We did verify however that the code correctly handled the callback.

Discussion. When Aldrich et al. re-engineered Aphyds to specify its architecture in ArchJava, they used `component class`es to create the hierarchy [6]. Here, we used public domains to create logical containment. The information we gleaned from the OOG is consistent with what Aldrich et al. found [6], but did not require re-engineering the application. There was a mismatch between the edges that the developer diagram intended to show and the ones that the OOG currently shows (field references). We plan to add control and data flow edges to the OOG.

7 Discussion

Evaluation. Our evaluation on these representative subject systems answers adequately our earlier research questions. We form the following hypotheses for future research, outlined in italics below. The requirements for a runtime architecture from Section 2.1 dictated some of the hypotheses. We also applied a taxonomy for software exploration tools by Storey et al. to runtime architectures instead of code architectures [47].

H1: The abstraction by types in an OOG effectively abstracts related instances. The developer controls the abstraction by selecting appropriate trivial types or design intent types.

H2: A developer can use the OOG to relate the runtime architecture and the code architecture and identify potential design flaws.

H3: The OOG hierarchy effectively abstracts instances compared to a raw object graph. Manually expanding or collapsing a few key substructures fine-tunes the visual representation.

H4: The OOG helps a developer learn how to refine the annotations to obtain the desired architectural view.

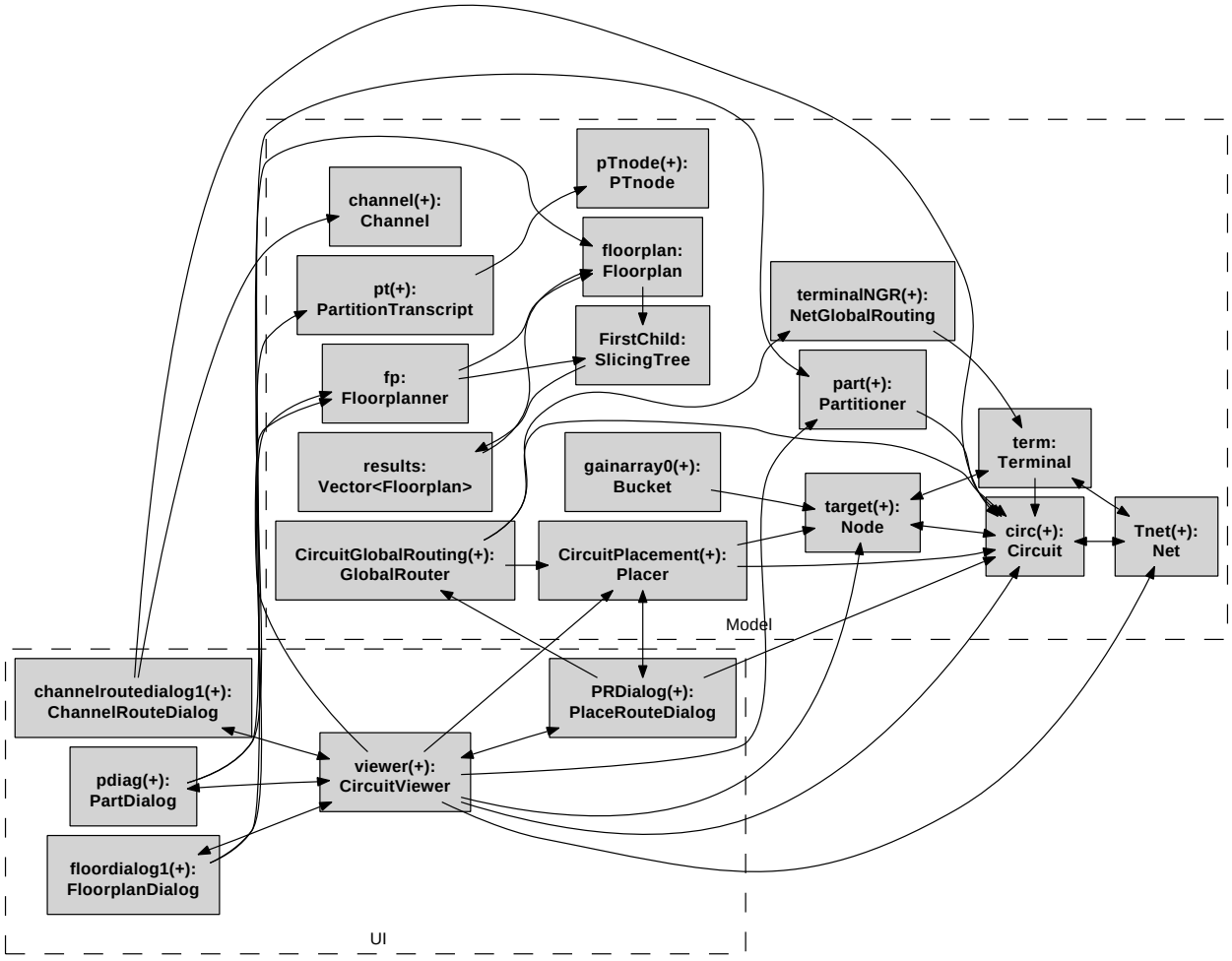


Figure 29: Aphyds OOG obtained with two top-level domains and many private domains. Note the (+) symbol on most objects to indicate that their substructure was elided.

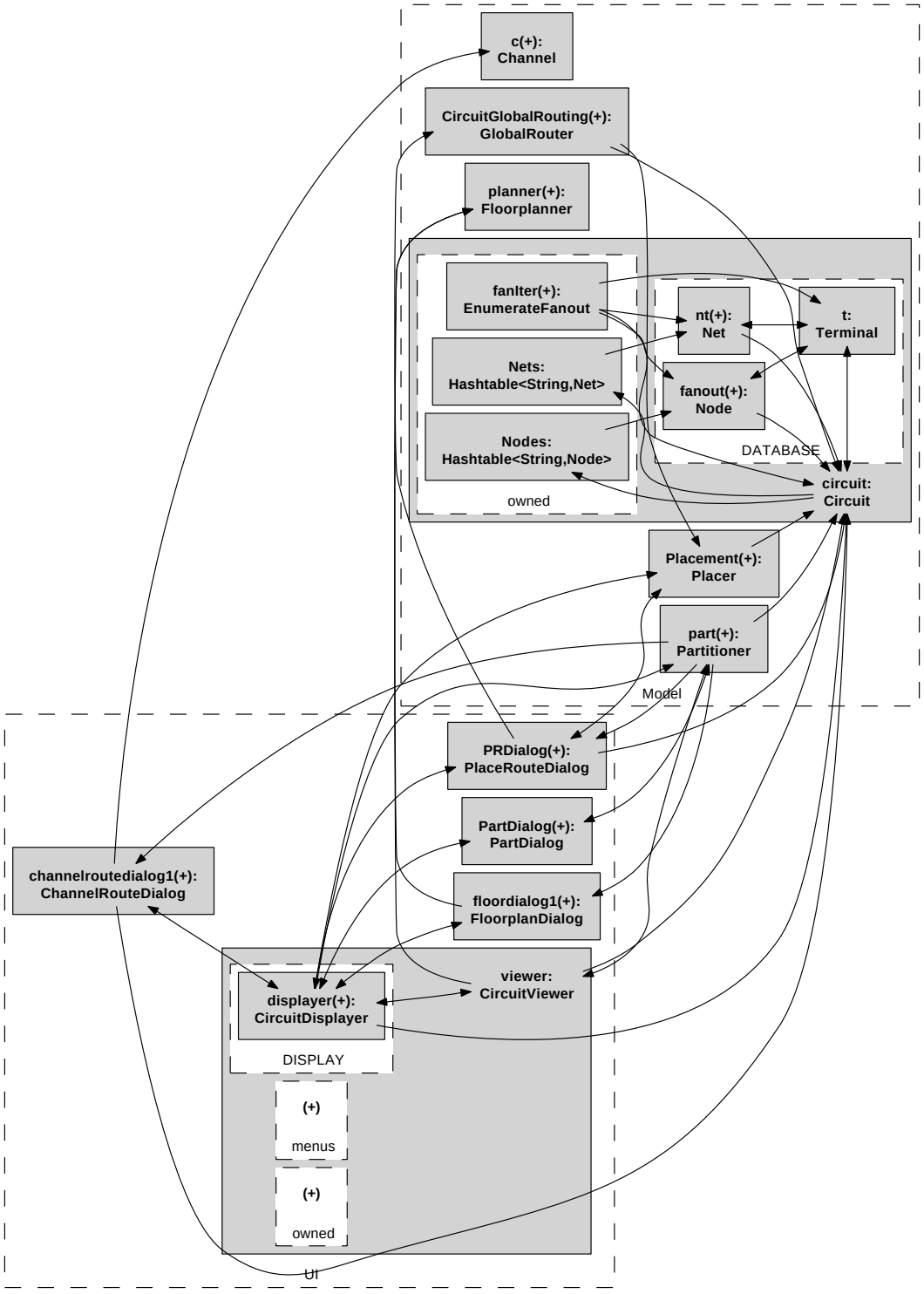


Figure 30: Aphyds OOG obtained by defining additional public domains.

Annotation Overhead. We currently add the ownership domains annotations manually so we used medium-sized programs for the evaluation. The annotation cost could be reduced by ownership inference [7, 28] and amortized over other uses of ownership, e.g., for program verification [27]. On the other hand, the annotations allow the developers to obtain meaningful architectural abstractions, rather than automatically inferred ones that are likely to be poor matches to the desired software architecture.

For the proposed approach to be adoptable, better ownership inference must be developed. Existing ownership inference tools adopt restrictive notions of ownership, do not map their results to a type system, do not infer ownership parameters [28] or infer imprecise ones [7].

ArchJava. The annotation burden is lower than re-engineering to ArchJava. Both authors previously re-engineered existing systems to ArchJava. ArchJava prohibits returning references to instances of "component classes" when most object-oriented code returns object references liberally. For Aphyds, Aldrich et al. converted dynamic connections into static ones and inadvertently injected several defects to produce a system with over 20 components and 80 "ports" in the code [6]. Aldrich et al. previously added ownership annotations to a subset of the Aphyds subject system (around 3,500 lines) in "three hours and 40 minutes — less than a quarter of the time that it took the same programmer to express the control-flow architecture of the same part of Aphyds" in ArchJava [6]. Similarly, when we re-engineered HillClimber, we initially introduced runtime exceptions due to disconnected ports and had to change the application's initialization order [4].

What are "good" annotations? Just as there are multiple architectural views of a system, there is no single right way to annotate a program. Good annotations minimize the number of top-level components in the OOG by pushing more objects underneath other objects. The best annotations produce an OOG that is comparable to what an architect might draw for the as-designed architecture, as long as the program supports them!

The OOG can guide a developer to refine the annotations. For JHotDraw, we initially placed `Handle` instances in the `CONTROLLER` domain and later moved them to the `Model` domain since `Handle` is related to `Figure`[12].

How is the Process Iterative? Obtaining an OOG is a semi-automated iterative process, which involves the following steps:

1. Decide on the right top-level domains;
2. Decide on the right top-level objects in the top-level domains. This may involve moving objects between the top-level domains;
3. Achieve an adequate number of objects in each top-level domain:
   (a) Push secondary objects underneath primary objects, using:
      i. Strict encapsulation (private domains);
      ii. Logical containment (public domains);
   (b) Pass low-level objects linearly between domains;
   (c) Use abstraction by types to merge fewer or more objects in each domain, using:
      i. Trivial types (Section 5.2): when there are many interfaces that are implemented by many types;
      ii. Design intent types (Section 5.3): when the design can be represented in terms of a few core types;
4. Achieve an appropriate level of visual detail:
   (a) Hide or show the substructure of a selected object;
   (b) Change the projection depth uniformly across objects;

   The tool adds any summary edges corresponding to the elided substructure.

Why was one system refactored? Adding the annotations to HillClimber highlighted refactoring opportunities [3, 4]. Without refactoring, the annotations would enforce a degraded architecture. We did not refactor JHotDraw or Aphyds, except to use generic types. These were code bases that were developed prior to Java 1.5. Most Java code is being refactored to use generics using available tool support [16].

[12] Ralph Johnson, who worked on the HotDraw precursor to JHotDraw, examined an earlier extracted OOG and recommended this change.

When adding annotations, one must choose between enforcing a degraded architecture or refactoring to reduce tight coupling, e.g., by programming to an interface, or introducing a mediator object [3]. E.g., in HillClimber, the mediator object was introduced during a refactoring.

OOG ≈ C&C View? We implemented a separate analysis to convert an OOG into a standard component-and-connector architecture. We also designed an analysis to compare the as-built C&C view obtained from the OOG to an as-designed architecture, and check their structural architectural conformance [1]. This paper focuses on architectural extraction.

Why Ownership Domains? The approach was presented in terms of the ownership domains type system, where each object contains one or more public or private domains, and each object is in exactly one domain. In principle, the approach also applies to ownership type systems that assume a single *context* per object [12]. However, in an owner-as-dominator type system, any access to a child object must go through its owning object [12]. In contrast, the ownership domains type system supports pushing 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 [5]. This expressiveness makes it possible to avoid an architecture that has too many top-level objects, as in the first Aphyds OOG (Figure 6.3). If making an object owned by another object restricts access to the owned object, this forces more objects to be peers.

Dynamism. The OOG is an approximation of the actual runtime architecture, one that is conservative and may include more than actually will be there by virtue of using a sound static analysis. However, the experimental evidence we have gathered on several extended examples, as well as many other smaller examples, indicates that the extracted architectures do not suffer from too much or too little abstraction. In comparison, Rayside et al. reported that a static object graph analysis based on RTA produced unacceptable over-approximations for most non-trivial programs [38]. Finally, the approach currently describes a static component-and-connector architecture of a system, but offers no facilities for specifying runtime architectural changes [34], as in dynamic architecture description languages. As a result, the approach does not address dynamic architectural reconfiguration [29]. In addition, our approach works for applications that run in a single virtual machine, so it handles neither heterogeneous nor distributed systems.

Performance. Table 1 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, build the abstract graph, convert it into a runtime graph, and then into a display graph. The IBV time is lower because when using the instantiation-based view, there are fewer abstract objects that the analysis must manipulate. Overall, the OOG tool is sufficiently interactive to allow iteration.

8 Related Work

Architectural Recovery. There is a large body of research on architectural recovery or architectural extraction. Most approaches use a mix of dynamic and static information such as naming conventions and directory structures [39]. The extractors often play detective and use trial and error with clustering

Table 1: **OOG** measures the extraction time on an Intel Pentium 4 (3 GHz) with 2 GB of memory. **WARN** is the remaining annotation warnings. **IBV** and **ABST** indicate if instantiation-based views or abstraction by types were used, respectively.

System	LOC	OOG	ABST	IBV	WARN
JHotDraw	15,000	2'18"	No	No	60
JHotDraw	15,000	0'16"	Yes	Yes	60
HillClimber	15,000	0'26"	No	No	42
HillClimber	15,000	0'09"	Yes	Yes	42
Aphyds	8,000	0'24"	No	No	72

algorithms [23]. Even so, existing compile-time approaches mostly obtain abstracted code architectures, not runtime architectures [11]. In many approaches, the abstraction mechanism is hard-coded in a tool and cannot be controlled with user-specified annotations.

Clustering methods are complementary to this approach and may help with annotating an unfamiliar system. For instance, if a clustering method derives how classes in a package interact with other classes in another package, this may suggest creating two top-level domains corresponding roughly to the two packages. A small cluster that interacts with almost all other packages may indicate a possible library or utility package, and often times, objects in such a package are `shared`.

Mapping source to high-level models. Murphy el al. produce a mapping of a source to a high-level model using the Reflexion Models (RM) approach [32]. In RM, the developer assigns component families to classes using an external file. Then the tool checks the relationships between these components and reports any differences to the developer. There are several important differences with RM. First, the object-oriented version of the RM method (embodied in the jRMTool) maps classes to components. Such a mapping is not appropriate for an object-oriented runtime architecture and is more suitable for the code architecture. A runtime architecture models runtime entities and their potential interactions. Thus, in an object-oriented system, a component is one or more objects. More specifically, RM cannot map the same code entity to multiple design elements, depending on the context of where they were used. A runtime view of an object-oriented system may distinguish between two instances of the same class in two different contexts. For instance, a `dataAccess` component may connect to a `settingsDB` component to read trusted configuration settings, and a `dataDB` component to access untrusted user data. A security analysis that operates on that runtime view may assign a High `trustLevel` for `settingsDB` and a Low `trustLevel` for `dataDB`. In contrast, a module view would show one element, assuming that the components are implemented as two instances of the same `java.io.File` class. RM can only map the java.io.File class to a single node in the high-level model. Second, RM does not extract a complete abstraction to avoid obtaining a model that developers do not recognize. In our method, the OOG represents a complete model, but developer-specified annotations help obtain meaningful abstractions. RM uses non-hierarchical high-level models and maps, whereas our method produces hierarchical representations. A developer writing the map manually must ensure that a type and its subtypes are mapped to the same entity in the high-level model. When mapping field or local variables, the developer must also ensure that all objects that may be aliased are mapped to the same high-level entity. In contrast, in our method, a type system checks that the annotations are consistent, and that the code is consistent with the annotations. And the construction of the runtime graph handles aliasing and inheritance. Producing the mapping file in the RM approach appears more straightforward than adding ownership annotations, but it is not amenable to type inference. The more sophisticated source abstraction method is needed to handle the runtime architectures of object-oriented systems soundly, in the presence of inheritance and aliasing.

Dynamic Analyses. There are several dynamic analyses for visualizing runtime structures [44, 15]. As mentioned earlier, a static analysis is often preferred to a dynamic analysis. First, runtime heap information does not convey design intent. Second, a dynamic analysis may not be repeatable, i.e., changing the inputs or executing different use cases might produce different results. Compared to dynamic ownership analyses — which are descriptive and show the ownership structure in a single run of a program, the OOG obtained at compile time is prescriptive and shows ownership relations that will be invariant over all program runs. Third, 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 [15], which must be incurred each time the analysis is run, whereas the main cost of adding annotations is incurred once.

DiscoTect [44] recovers a non-hierarchical C&C view from a running program, one that shows one component for each instance created at runtime. Such views must be manually post-processed to consolidate multiple components into one. An automated analysis can convert an OOG into a C&C view that is hierarchical and does not require manual post-processing [1].

Visualization. Software visualization research shows different aspects of the execution structure of a running program [45, 14]. Compared to our approach, dynamic visualization approaches do not require source

code annotations and allow more fine-grained user interaction in producing abstractions. But these analyses often work at the granularity of an object or a class and produce task-specific views. Our contribution in this paper is not a visualization, it is in having developer-specified ownership annotations drive a sound static extraction of a system's runtime architecture.

Dynamic Ownership Analyses. More closely related are dynamic analyses that infer the runtime ownership structures. These techniques do not require program annotations but assume a strict owner-as-dominator model which cannot represent many design idioms.

Rayside et al. [37] produce matrix displays of the ownership structure. Similarly, Mitchell [31] uses lightweight ownership inference to examine a single heap snapshot rather than the entire program execution, and scales the approach to large programs through extensive graph transformation and summarization. Noble, Potter, Potanin et al. showed both matrix and graph views of ownership structures [20, 35] and demonstrated that ownership is effective at organizing runtime object structures. We use the same key insight but in a static analysis that addresses additional challenges.

Object Graph Analyses. Several static analyses produce non-hierarchical object graphs without using annotations. PANGAEA [46] produces a flat object graph without an alias analysis and is unsound (the PANGAEA output for JHotDraw is even more complex than Figure 2). WOMBLE [21] uses syntactic heuristics and abstraction rules for container classes to obtain an object model including multiplicities. The WOMBLE analysis is unsound and aliasing-unaware by design. AJAX [33] 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. Its output was manually post-processed to remove "lumps" with more than seven incoming edges [33, p. 248]. In our approach, we often suppress `shared` objects and their associated edges since they often add needless clutter. Even though excluding `shared` objects makes the resulting architecture unsound, the use of the `shared` annotation is entirely under the control of the developer adding the annotations. A developer can avoid the `shared` annotation if she is interested in reasoning about all the objects in the system: `shared` is an escape hatch mainly designed to easily inter-operate with legacy code or third-party libraries [7]. Moreover, AJAX's heavyweight but precise alias analysis does not scale to large programs. Flat objects graphs do not provide architectural abstraction and do not scale, because the number of top-level objects in the architecture increases with the program size.

Lam and Rinard [25] proposed a type system and a static analysis (which we 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 two kinds of tokens. The first kind, token parameters, are a loose adaptation of ownership type parameters that predate them [12], and correspond roughly to domain parameters. Compared to Reflexion Models, LR can map a single code element to multiple design elements. Token parameters lack semantics, i.e., they do not give any precision about aliasing. The second kind, global static tokens correspond loosely to top-level domains. In LR, each token parameter $C<p_1>$ is bound to another $B<p_2>$, and transitively to a global token. In ownership domains, a class C can declare a private or a public domain D. Each instance of C gets a fresh instance of D, so $obj_1.D \neq obj_2.D$ for fresh obj_1 and obj_2. In addition to binding to another domain parameter $B<d_2>$, a domain parameter $C<d>$ can bind a locally declared domain, e.g., $objB.D$. These local domains create the OOG hierarchy. As a result, an analysis based on LR can only extract non-hierarchical object models. For instance, for Aphyds, we used private domains in Round1 – note the (+) sign on most objects for the elided substructure in Figure 6.3. To get the OOG in Figure 6.3, one that is comparable to the as-designed architecture in Figure 1, we used public domains in Round 2, thus confirming that hierarchy is indispensable. Lam and Rinard do not mention inheritance in their paper and their formal system omits inheritance entirely [25, Fig. 10]. This paper discussed the challenges that multiple interface inheritance introduces. Lam and Rinard gave no soundness proof of the underlying type system or of the extracted object model. Finally, the LR system was evaluated on one 1.7KLOC system, whereas we evaluated the OOG extraction on several systems of 8-15KLOC each. Unlike the ownership domains type system, the LR type system is only descriptive and does not enforce a tiered architecture in code [5, 3]. Our approach does not require special annotations just to extract a design [25] but leverages well-researched ownership types [12, 5] that also have uses in program verification [27]. Finally,

work on ownership inference could reduce the annotation cost [7, 26, 28].

Similarly, a *package* in confined types [10], which track classes not instances, can be considered a package-level static ownership domain and thus coarser than a token. However, confined types carry a lower syntactic overhead than ownership types.

Shape Analysis. Our analysis creates a graph that summarizes possible relationships among objects at runtime. Shape analysis, e.g., [41], is related but differs on two counts. First, shape analyses are whole-program analyses that do not scale. Second, they produce heap abstractions that show a graph consisting of nodes to represent a set of objects and edges to represent points-to relations. Our representation is hierarchical, whereby a set of objects is contained inside a domain of another object. Hierarchy allows varying the level of architectural abstraction (See Fig. 6(c)). 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 keep as separate two objects that are in distinct domains, because the underlying type system guarantees they can never alias.

9 Conclusion

We proposed a novel approach to statically extract a sound hierarchical object graph from an object-oriented program that is written in an existing language and that uses existing libraries and general design idioms. The approach relies on ownership domain annotations to specify and enforce in code the architectural intent related to object encapsulation and communication.

We evaluated the approach using several real medium-sized programs. From an annotated program, a tool can quickly extract a runtime architecture that conveys meaningful abstractions and gives various insights by identifying undocumented information or contradicting manual documentation. In follow-up work, we convert an extracted object graph into a standard component-and-connector architecture and check its conformance with an as-designed architecture [1]. In future work, we plan to run architectural-level analyses for properties such as performance or security on an extracted runtime architecture.

Acknowledgements

The authors would like to acknowledge the PLAID group for their continuous feedback. We also thank Robert J. Simmons for his helpful comments.

References

[1] M. Abi-Antoun and J. Aldrich. Checking and Measuring the Architectural Structural Conformance of Object-Oriented Systems. Technical Report CMU-ISRI-07-119, Carnegie Mellon University, 2007.

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

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

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

[5] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[6] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *ICSE*, 2002.

[7] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.

[8] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *OOPSLA*, 1996.

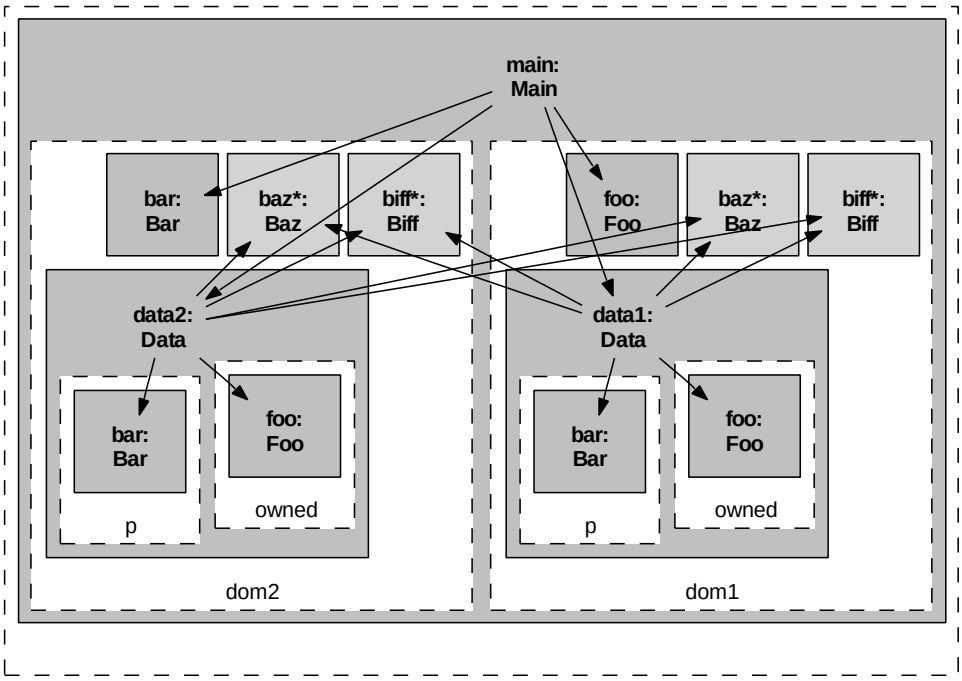
[9] J. Bloch. *Effective Java*. Addison-Wesley, 2001.

[10] B. Bokowski and J. Vitek. Confined Types. In *OOPSLA*, 1999.
[11] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a Case Study: its Extracted Software Architecture. In *ICSE*, pages 555–563, 1999.
[12] D. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.
[13] P. Clements et al. *Documenting Software Architecture*. Addison-Wesley, 2003.
[14] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the Execution of Java Programs. In *Software Visualization*, 2002.
[15] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *FLoC FATES-RV*, 2006.
[16] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *ECOOP*, 2005.
[17] E. Gamma. Advanced Design with Patterns and Java (Tutorial). In *JAOO*, 1998. JHotDraw version 5.1.
[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[19] E. R. Gansner and S. C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Softw. Practice & Exp.*, 30(11), 2000.
[20] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Visual Languages and Computing*, 13(3), 2002.
[21] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.
[22] W. Kaiser. Become a Programming Picasso with JHotDraw. JavaWorld, 2001.
[23] R. Kazman and S. J. Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Softw. Eng.*, 6(2), 1999.
[24] H. Koning, C. Dormann, and H. van Vliet. Practical Guidelines for the Readability of IT-Architecture Diagrams. In *SIGDOC*, 2002.
[25] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.
[26] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *ICSE*, 2007.
[27] Y. Lu, J. Potter, and J. Xue. Validity Invariants and Effects. In *ECOOP*, 2007.
[28] K.-K. Ma and J. S. Foster. Inferring Aliasing and Encapsulation Properties for Java. In *OOPSLA*, 2007.
[29] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *FSE*, pages 3–14, 1996.
[30] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE*, 26(1), 2000.
[31] N. Mitchell. The Runtime Structure of Object Ownership. In *ECOOP*, 2006.
[32] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE TSE*, 27(4), 2001.
[33] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, CMU, 2001.
[34] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *ICSE*, 1998.
[35] A. Potanin, J. Noble, and R. Biddle. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience*, 16(7), 2004.
[36] J. Potter, J. Noble, and D. Clarke. The Ins and Outs of Objects. In *Australian Softw. Eng. Conf.*, 1998.
[37] D. Rayside, L. Mendel, and D. Jackson. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Workshop on Dynamic Analysis (WODA)*, 2006.
[38] D. Rayside, L. Mendel, R. Seater, and D. Jackson. An Analysis and Visualization for Revealing Object Sharing. In *Eclipse Technology eXchange (ETX)*, 2005.
[39] T. Richner and S. Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In *ICSM*, 1999.
[40] D. Riehle. *Framework Design: a Role Modeling Approach*. PhD thesis, ETH Zurich, 2000.
[41] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *POPL*, pages 105–118, 1999.
[42] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using Dependency Models to Manage Complex Software Architecture. In *OOPSLA*, 2005.
[43] J. Schäfer, M. Reitz, J.-M. Gaillourdet, and A. Poetzsch-Heffter. Linking Programs to Architectures: An Object-Oriented Hierarchical Software Model based on Boxes. In *The Common Component Modeling Example: Comparing Software Component Models*, 2008.
[44] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *IEEE TSE*, 32(7), 2006.
[45] M. Sefika, A. Sane, and R. Campbell. Architecture Oriented Visualization. In *OOPSLA*, 1996.
[46] A. Spiegel. *Automatic Distribution of Object-Oriented Programs*. PhD thesis, FU Berlin, 2002.
[47] M. Storey, F. Fracchia, and H. Müller. Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration. *J. Systems and Software*, 44(3), 1999.

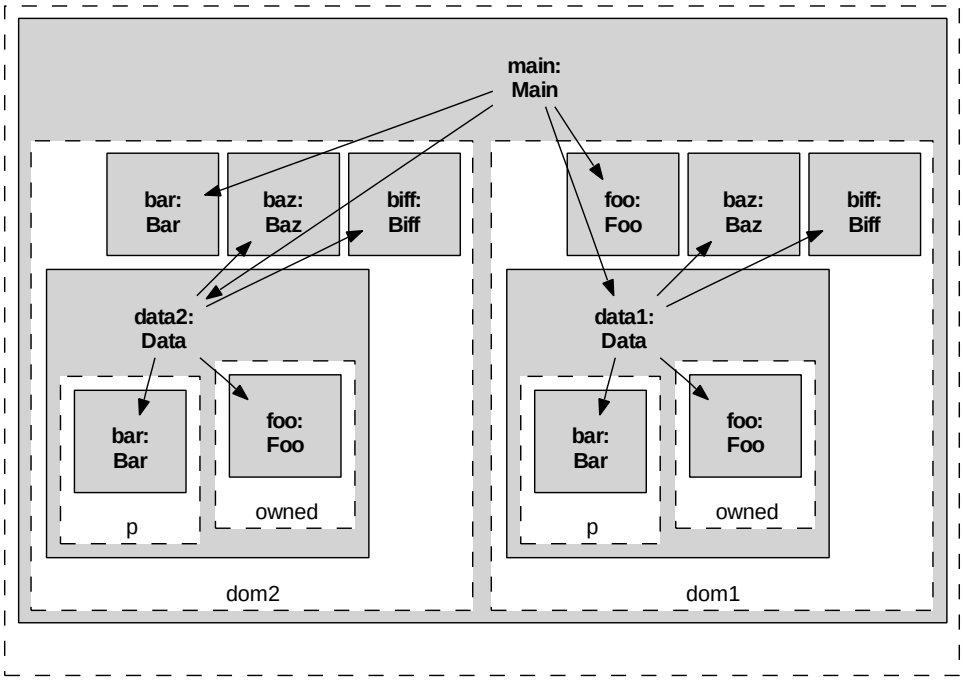
APPENDIX

A Edge Imprecision

The earlier pseudo-code for the OOG [2] generated imprecise edges (Figure 31), i.e., the OOG showed statically edges that do not really exist at runtime. While still sound, these extra edges needlessly clutter the OOG. We changed the data structures we use and the pseudo-code for adding edges to fix this imprecision.



(a) OOG with imprecise edges from `dom1.data1` to `dom2.biff` and `dom2.baz`.



(b) OOG without the imprecise edges.

Figure 31: An illustration of possible imprecision in the edges to "pulled objects" in the OOG.