Static Extraction of Sound Hierarchical Runtime Object Graphs

Marwan Abi-Antoun   Jonathan Aldrich

School of Computer Science, Carnegie Mellon University

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

Abstract

For many object-oriented systems, it is often useful to have a runtime architecture that shows networks of communicating objects. But it is hard to statically extract runtime object graphs that provide architectural abstraction from existing programs written in general purpose languages, and that follow 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 objects 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 68 KLOC of real code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

General Terms Experimentation, Languages, Theory

1. Introduction

Architects use different architectural views to describe a system. A *code architecture* organizes code entities in terms of classes, packages, layers and modules. A *runtime architecture* of a system models runtime entities and their potential interactions [9]. For an object-oriented system, a runtime architecture shows networks of communicating *components* that are aggregates of objects.

Intuitively, many have preferred dynamic analyses to extract runtime architectures [10]. But, a dynamic analysis extracts, from a few program runs, partial descriptions that cover interactions between objects. Ideally, an architecture must capture a complete description of a system's runtime structure. This requires a static analysis that is *sound*, one that reveals all objects and relations that may exist at runtime. Several static analyses have been proposed, some sound with respect to aliasing [18, 14], others not [12, 26]. But previous static approaches, whether sound or unsound, extract low-level non-hierarchical object graphs that do not provide architectural abstraction and do not scale to large programs.

TLDI'09 January 24, 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM 978-1-60558-420-1/09/01…$5.00
Reprinted from TLDI'09, Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation, January 24, 2009, Savannah, Georgia, USA., pp. 1–**??**.

To simplify the problem, some approaches radically change a language's type system to include architectural constructs [5, 24] or mandate specific implementation frameworks. An approach that handles existing systems must support existing languages, common design idioms and existing frameworks and libraries.

The key insight of our work is to leverage ownership types for architectural extraction. Ownership types were originally proposed to control aliasing [8, 4]. But the ideas and techniques of ownership seem crucial for extracting sound runtime architectures at compile-time. Ownership types track instances instead of classes, and a runtime architecture shows objects.

Our principled architectural recovery combines type annotations and a type-based static analysis. A developer guides the architectural abstraction by adding annotations to the source code to clarify the architectural intent. The annotations specify in code object encapsulation, logical containment and architectural tiers, which are not explicit constructs in general purpose programming languages. A static analysis then extracts sound hierarchical object graphs that provide architectural abstraction, first by ownership hierarchy, and then by types.

Lam and Rinard previously proposed using annotations to recover object models from code [14], but their system did not support hierarchy, and thus cannot scale to large systems at multiple levels of abstraction, nor did their paper discuss critical language constructs like inheritance.

Our approach does have the overhead of adding the annotations to a program, which we currently do mostly manually. Precise and scalable ownership inference is a separate problem and an active topic of ongoing research [15, 16]. Our contribution in this paper is a static analysis for extracting sound hierarchical runtime object graphs based on well-understood ownership type annotations. We proved soundness results for the core of the technique and evaluated it on 68 KLOC of real object-oriented code.

This paper is organized as follows. We define runtime architectures (Section 2) and motivate the use of annotations. In Section 3, we describe the core analysis informally. In Section 4, we describe the analysis formally and prove key soundness theorems. In Section 5, we discuss additional architectural abstraction by types. We conclude with a discussion and related work (Section 6).

2. Motivation

A *Runtime Object Graph (ROG)* represents the runtime structure of an object-oriented program. Nodes correspond to runtime objects. Edges correspond to relations between objects, such as field points-to reference relations, or usage relations such as method invocations. Our static analysis constructs a representation that soundly approximates any ROG that any program run may generate.

Different executions generate a different number of objects. Furthermore, the number of objects in a ROG is unbounded. An architecture must be a finite representation of a ROG, whereby one architectural component represents multiple objects at runtime.

Existing static analyses that extract a system's runtime structure produce low-level non-hierarchical object graphs that explain runtime interactions in detail [18, 12, 26], but convey little architectural abstraction (see examples in the companion report [3]). In particular, low-level objects appear at the same level as the architecturally relevant objects, and there is no way to distinguish them.

A static analysis must also handle possible aliasing. Ignoring aliasing may produce a misleading object graph. If two object references could alias, a sound analysis must map them to the same architectural component. If an architecture showed two components for one runtime entity, an analysis at the architectural-level may assign these two components different values for a key architectural property, which could invalidate the results of the analysis.

We listed earlier several requirements on a solution, namely that it be a static analysis and not require language extensions. We define the following key properties of a runtime architecture.

In a runtime architecture, a *component* is an object or a group of objects [9]. An architecture also represents the relations between an object in one component and some other object in another component. An architecture is often hierarchical and decomposes each component into a nested sub-architecture.

An architecture often organizes components into runtime *tiers*, where a *tier* is a conceptual partitioning of functionality, for example, to distinguish presentation from computational elements.

An architecture *scales* if the top-level architecture stays mostly the same as the program size increases arbitrarily. An architecture is *sound* if it represents all objects and relations between objects that may exist at runtime. Finally, an architecture is *precise* if two runtime entities that represent different conceptual design elements appear as different architectural entities.

2.1 Mapping from Source to High-Level Models

One way to view architectural extraction is that it maps code elements to elements in a high-level architectural model. 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. The code for this example is in Fig. 1, where the annotations are shown as underlined. We will explain the annotations in Section 2.2.

We indicate that tier D contains component C using the notation $D{::}C$. 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), as follows:

```
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. 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:

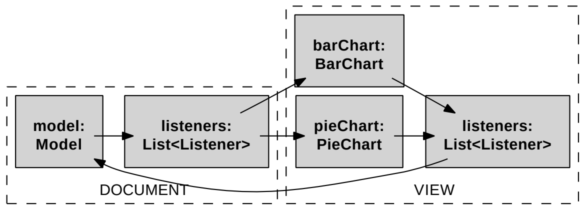
```
object System.barChart to VIEW::barChart
object System.model to DOCUMENT::model
```

In Fig. 2(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:

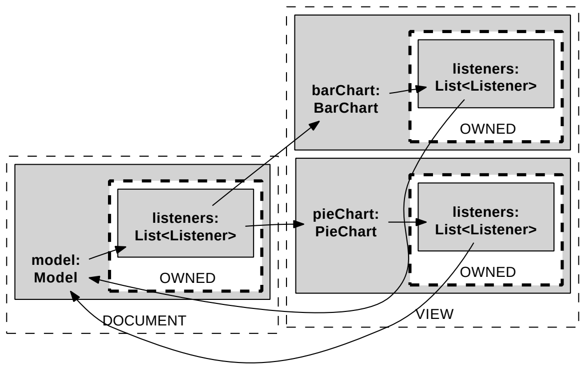
```
object BarChart.listeners to VIEW::listeners
object Model.listeners to DOCUMENT::listeners
```

```
interface Listener {
}
class BaseChart<M> // Declare domain parameter M
      implements Listener {
  domain OWNED; // Declare protected domain OWNED
  // Declare reference to List object in OWNED
  // Inner annotation M is for the list elements
  OWNED List<M Listener> listeners;
}
class BarChart<M> extends BaseChart<M> {
}
class PieChart<M> extends BaseChart<M> {
}
class Model<V> implements Listener {
   domain OWNED;
   // Inner annotation V is for the list elements
   OWNED List<V Listener> listeners;
}
class Main {
  domain DOCUMENT, VIEW; // Top-level domains
  // Bind domain parameter V to actual domain VIEW
  DOCUMENT Model<VIEW> model;
  VIEW BarChart<DOCUMENT> barChart;
  VIEW PieChart<DOCUMENT> pieChart;
}
class List<ELTS T> { // ELTS is domain for the List elements
                   // T is a generic type parameter
  ELTS T obj; // "Virtual field" to summarize implementation
}
```

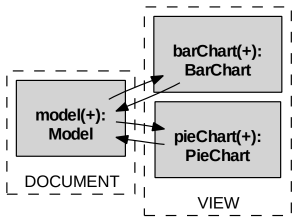
Figure 1. Document-View system with annotations.



(a) Flat runtime architecture.



(b) Hierarchical runtime architecture using nested boxes.



(c) Eliding the sub-structures.

Figure 2. Runtime architecture of Document-View system.

```
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 }
```

Figure 3. DataAccess code with annotations.

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. 2(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 [14]. Our system uses a similar solution. 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 a 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 substructure of an object is elided (Fig. 2(c)).

2.2 Ownership Domains

In our approach, a developer guides the architectural abstraction by adding annotations to clarify the architectural intent in the code. The annotations make the extracted architecture reflect a developer's architectural intent rather than a tool's heuristic.

The annotations assign each object to a single *ownership domain* that does not change at runtime. An ownership domain is a *conceptual group of objects* with an explicit name and explicit policies that govern how it can reference objects in other domains [4].

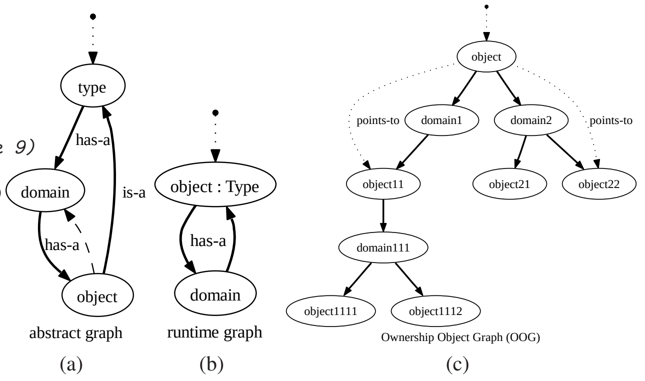


Figure 4. Transformation between abstract graph, runtime graph and Ownership Object Graph (OOG). The OOG is for illustration purposes and is not produced from a particular runtime graph.

Each class can declare one or more *public* or *private* domains to hold its internal objects, thus supporting hierarchy. Although a domain is declared at the level of a class in a program, each instance of that class has its own runtime domain. In particular, an annotation can refer to the public domain D of an object o, similarly to a field access, as $obj.D$. Whenever our analysis distinguishes two objects obj_1 and obj_2, it also distinguishes the domains that these objects contain in turn, such as $obj_1.D$ and $obj_2.D$.

An object x of type X can access objects in a domain D of object y of type Y by declaring a formal *domain parameter* F on X and *binding* F to domain D. Objects inside a private domain are encapsulated. Permission to access an object implies permission to access its public domains. *Domain links* describe allowed object communication [4], but do not introduce new difficulties compared to computing object relations, so we do not discuss them further.

Example. Fig. 3 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. Domain names are arbitrary, except for a few special annotations [4]. We often use capital letters to distinguish them from other program identifiers.

Class `DataAccess` declares a public domain `STATE` (Line 2), and `Integer` and `Number` objects in `STATE` (Lines 3,4). Any object that has a reference to a `DataAccess` object can access the objects in its public domain. In addition, `DataAccess` requires some environment's state that it does not own, so it declares a domain parameter `PENV` (Line 1). An object of type `UnitTest` declares an `ENV` domain (Line 12), and binds it to `DataAccess`'s `PENV` (Line 15), so that `DataAccess` can refer to `UnitTest`'s state. Similarly, `ArrayList` has a domain parameter `ELTS` for the list elements (not shown). The outer `PENV` annotation is for the `ArrayList` object itself. The inner `PENV` annotation binds `ELTS` to `PENV` (Line 9), to make the `ArrayList`'s `Integer` objects accessible in `PENV`.

3. Analysis

The analysis builds an *abstract graph*, converts it into a *runtime graph*, which is a sound representation of any runtime object graph (ROG) (See Fig. 4). We often do not display a runtime graph directly, as we discuss later, but instead display its projection, which we call an Ownership Object Graph (OOG).

3.1 Abstract Graph

An *abstract graph* represents the type structure of the objects that the code manipulates. A visitor builds an abstract graph from the

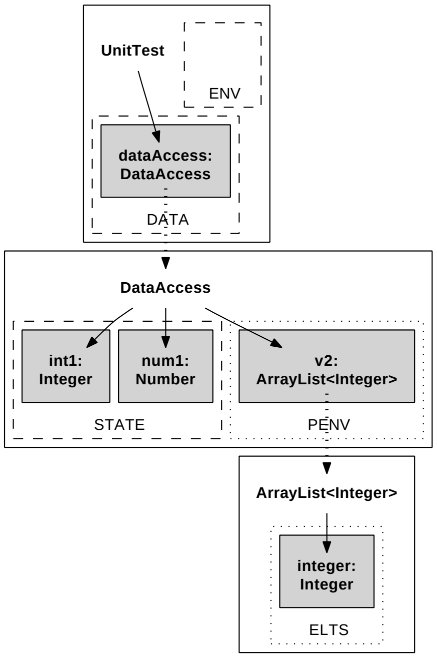


Figure 5. The *abstract graph* for `DataAccess`.

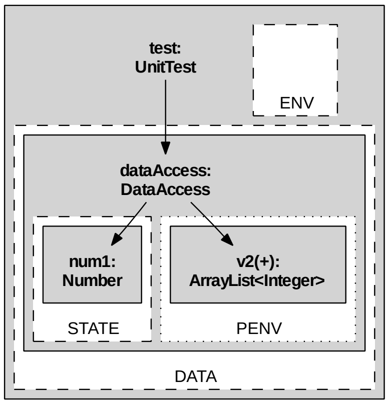
Abstract Syntax Tree (AST) of an annotated program and accounts for inheritance (Fig. 4(a)). An *abstract graph* has *abstract types*, and an *abstract type* has *abstract domains*, as declared in the program. An abstract domain has *abstract objects* corresponding to the fields and variables that the program declares in the domain.

Fig. 5 shows the abstract graph for the `DataAccess` system. A white-filled solid-border box represents an abstract type. A white-filled dotted-border box represents a formal domain parameter, e.g., `PENV`, declared inside a type. A white-filled dashed-border box represents an actual domain, e.g., `STATE`. A grey-filled box represents an abstract object declared inside a domain. A thick dotted edge represents a type relationship. A solid edge represents a field reference.

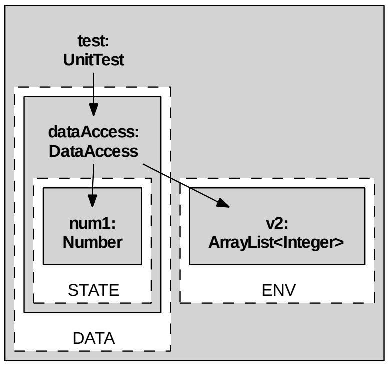
An abstract graph is inadequate as an architecture. An abstract graph is not hierarchical in the sense of an object having children. Rather, an abstract object has an abstract type, an abstract type has abstract domains, and an abstract domain has abstract objects. For example, abstract object `dataAccess` has type `DataAccess`, and abstract type `DataAccess` has domains `STATE` and `PENV`, and abstract domain `STATE` contains the abstract object `int1:Integer`.

An 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 of compatible types, in the same domain, may alias. E.g., `int1` and `num1` in `STATE` may refer to the same object because `Integer` is a subtype of `Number`.

If two objects may alias, a runtime architecture must show them as one. In general, merging objects based on only the aliasing precision that the ownership domains type system provides could yield imprecise results. For example, one could use an intra-domain alias analysis to better approximate the set of objects that may alias at runtime. But our experience in applying the analysis on 68 KLOC of real code confirms that the annotations give more than enough precision about aliasing, as long as most object references are declared — or instantiated — with precise types, instead of `java.lang.Object` (See Section 5.1). In fact, we often need additional abstraction by types in a domain, as we discuss in Sections 5.2–5.3. In practice, to avoid merging all objects in a domain that have a raw type, e.g., `Vector`, we suggest but do not require refactoring the code to use a generic type, e.g., `Vector<T>`.



(a) Runtime graph *before* pulling.



(b) Runtime graph *after* pulling.

Figure 6. Partial `DataAccess` runtime graphs.

An abstract domain in an abstract graph does not directly show all the objects that are 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 fields do not appear where the actual domain is declared. Hence, in the abstract graph, domain `ENV` inside `UnitTest` is empty (Fig. 5).

The analysis converts an *abstract graph* into a *runtime graph*, which soundly approximates any true runtime object graph (ROG).

We adopt the following graphical conventions (Fig. 7). A dashed (dotted) border white-filled rectangle represents an actual (formal) ownership domain, respectively. 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 $(+)$ on an object's label indicates that its substructure is elided.

3.2 Runtime Graph

A *runtime graph* instantiates the types in an abstract graph, as possibly different runtime objects in different domains, and shows only runtime objects and domains. Each *runtime object* contains *runtime domains* and each *runtime domain* contains *runtime objects*. Thus, in a runtime graph, one can view the children of an object without going through its declared type.

At a high level, the analysis distinguishes between objects in different domains, and abstracts runtime objects to pairs of domains and types. The analysis adopts the following approach to possible aliasing: in a given domain, two abstract objects with compatible types are merged. The analysis also eliminates formal domain

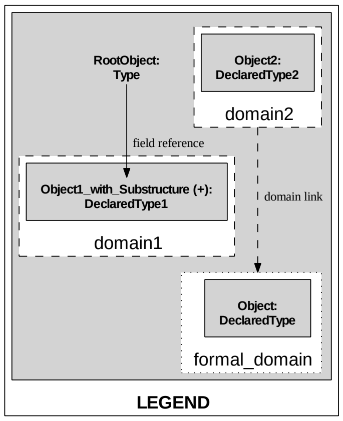


Figure 7. Legend.

parameters by substituting them with actual domains. Finally, the analysis adds edges between objects. We now discuss each in turn.

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

For instance, a dynamic analysis might show individual cells in a linked list of `Integer` objects, such as `cons1:Cons`, `cons2:Cons`, among others. Our approach unifies all the `Cons` cells into one `cons:Cons` object and a self-edge represents the reference from a cell to the next one (Fig. 9(a)).

Object Aliasing. When converting abstract objects from an 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 `DataAccess`, up until this point, is in Fig. 6(a). One runtime object, labeled with `num1:Number`, merges the abstract objects `int1` and `num1` in domain `STATE` (`Integer` is a subtype of `Number`).

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

Fig. 6(a) shows object `v2` in the formal domain parameter `PENV`. In Fig. 6(b), object `v2` is pulled from the formal domain parameter `PENV` to the actual domain `ENV` in `UnitTest`. Unless the user requests otherwise, we elide formal domains after pulling, so Fig. 6(b) no longer displays `PENV`. Similarly, an `ArrayList<Integer>` object has a domain parameter `ELTS` (not shown) that contains `Integer` objects. The inner `PENV` annotation on `v2` binds `ELTS` to `PENV`. The analysis transitively pulls the `Integer` objects from `ELTS` into `PENV`, then into `ENV` (Fig. 8).

Object Relations. The solid edges in Fig. 6(b) correspond to field reference edges. 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, we could add usage edges that show field accesses or method invocations.

3.3 Ownership Object Graph (OOG)

In the presence of recursive types, a runtime graph may grow arbitrarily deep. Consider a class `QuadTree`, which declares fields of type `QuadTree` in its `OWNED` domain, as follows:

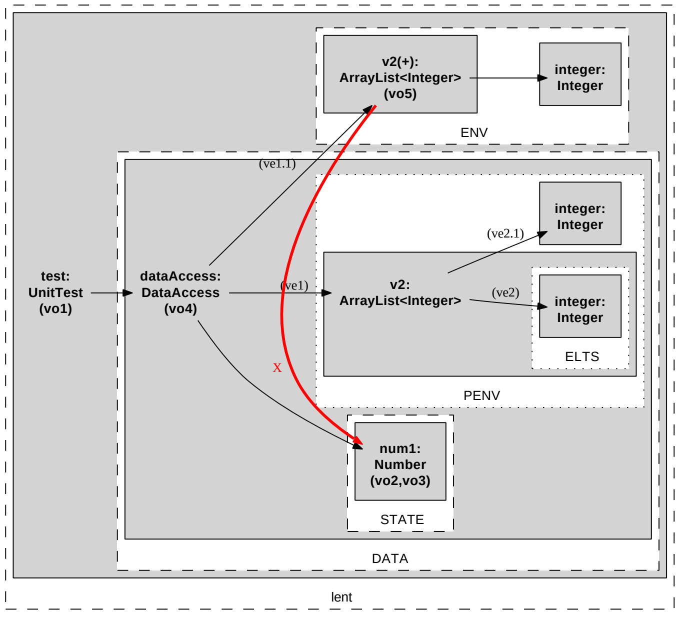


Figure 8. OOG showing formal domains and root object.

```
class QuadTree {
  domain OWNED;
  OWNED QuadTree _nwQuadTree;
  ...
}
```

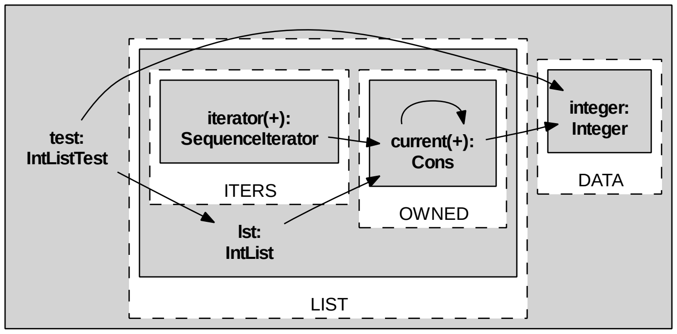
To ensure the analysis terminates, we want the runtime graph to be finite. So we create a unique canonical runtime object for each type in each domain declared in the program. Therefore, the object representing `QuadTree` in domain `OWNED` must also represent the child object of type `QuadTree` in the `OWNED` domain of the parent; it is therefore its own parent in this representation. To display a hierarchy in which no object is its own parent, the analysis creates an OOG as a finite depth-limited unrolling of the runtime graph. For the `QuadTree` example, we show one `QuadTree` object within another, down to a finite depth (see the middle of Fig. 15).

Edge Summaries. An OOG is depth-restricted but must still show all relations that exist at runtime. Merely truncating the recursion in a runtime graph 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 adds summary edges from the parent objects to those external objects.

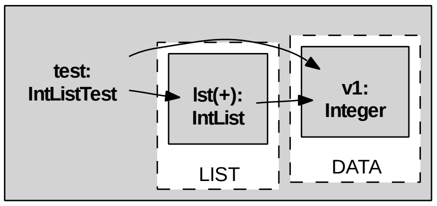
An OOG includes, for any two objects A and B that have an edge between them in a further unfolding of the runtime graph, an edge between objects A' and B', where A' is the object from which A is unfolded and B' is the object from which B is unfolded.

If the user-specified projection depth is d, the analysis projects the runtime graph to a depth of $d + n$ in order to produce summary edges that are due to the nodes below the cutoff depth. Although the depth of an actual runtime object graph is unbounded, our OOG is finite. For any runtime graph, we conjecture there is a fixed depth n that is sufficient to produce all such summary edges, and thus to produce a sound OOG. However, we have neither a proof nor an algorithm for computing n. Our soundness proof applies to the runtime graph, which does not include summary edges.

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. After pulling, a `Cons` object in domain `OWNED` refers to an `Integer` object in domain `DATA` that contains the list elements (Fig. 9(a)). If the projection depth is reduced to



(a) OOG for an `IntList` object.



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

Figure 9. Summary edges added after limiting the projection depth or hiding `lst`'s substructure.

elide `IntList`'s substructure, the analysis adds a summary edge from `IntList` to the pulled `Integer` object (Fig. 9(b)).

Recapitulation. An OOG is a graph with two types of nodes, objects and domains. The nodes form a hierarchy where each object node has a unique parent domain and each domain node has a unique parent object. The root of the graph is a top-level domain. There are two edge types. Edges between objects correspond to field references or usage relations. Edges between domains correspond to domain links, which we do not discuss here. Compared to earlier definitions of object graphs [20], an OOG explicitly represents clusters of objects using domains and edges between these clusters using domain links. In contrast to other ownership hierarchies [11, 19], in an OOG, the owner of an object is a domain instead of another object.

The root object of an OOG is often an instance of a class that only declares the top-level domains and the objects inside them. For readability, we sometimes elide the root domain and the root object from an OOG and consider the domains inside the root type as the top-level domains.

To provide architectural abstraction, an object graph must distinguish between objects that architecturally relevant and those that are not. An OOG provides architectural abstraction primarily by ownership hierarchy. It contains low-level objects into high-level architectural objects. Thus, only architecturally relevant objects appear in the top-level domains. Each of those objects has nested domains and objects representing subcomponents, and so on, until low-level less architecturally relevant objects are reached.

Collapsing many nodes into one is a classic approach to shrink a graph. However, an OOG collapses nodes based on the ownership structure, not according to where the program declares the objects, a naming convention or a graph clustering algorithm [13].

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.

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. 10).

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. The RuntimeGraph directly includes all the AbstractDomains, so 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, as we discuss later, 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.

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 [4], which follows standard nominal subtyping, and in addition, checks that all domain parameters are invariant with subtyping.

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 AbstractObject 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 [1].

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 Abstrac-

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

Object Rules $\boxed{\Theta \implies \Theta'}$

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

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

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

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

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

Auxiliary Rules

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

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

Edge Rules $\boxed{\Theta; \mathsf{RuntimeObject} \vdash \Omega \implies \Omega'}$

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

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

$$\frac{\mathsf{RuntimeEdge}([d_{src_1}, d_{src_2}\ldots], t_{src}, [d_{dst}\ldots], t_{dst}) \in \Omega \qquad \mathsf{RuntimeObject}(\{o\ldots\}, d) \vdash \mathsf{mapFtoA}(d_{src_1}, d_{src_2}) = d_{src'}}{\Theta;\ \mathsf{RuntimeObject}(\{o\ldots\}, d) \vdash \Omega \implies \Omega,\ \mathsf{RuntimeEdge}([d_{src'}\ldots], t_{src}, [d_{dst}\ldots], t_{dst})}\,[\text{R-SUBST-EDGE-L}]$$

$$\frac{\mathsf{RuntimeEdge}([d_{src}\ldots], t_{src}, [d_{dst_1}, d_{dst_2}\ldots], t_{dst}) \in \Omega \qquad \mathsf{RuntimeObject}(\{o\ldots\}, d) \vdash \mathsf{mapFtoA}(d_{dst_1}, d_{dst_2}) = d_{dst'}}{\Theta;\ \mathsf{RuntimeObject}(\{o\ldots\}, d) \vdash \Omega \implies \Omega,\ \mathsf{RuntimeEdge}([d_{src}\ldots], t_{src}, [d_{dst'}\ldots], t_{dst})}\,[\text{R-SUBST-EDGE-R}]$$

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

Figure 10. Data type declarations and rewriting rules to convert an AbstractGraph into a RuntimeGraph.

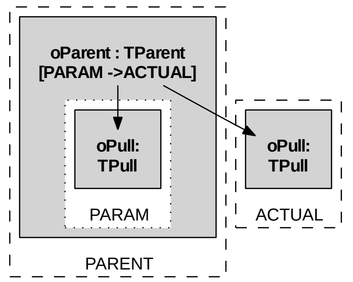


Figure 11. o_{pull} gets pulled from $PARAM$ to $ACTUAL$.

T-Store

$$\frac{\ldots \quad S[\ell] = C{<}\overline{\ell'.n}{>}(\overline{v}) \iff \Sigma[\ell] = C{<}\overline{\ell'.n}{>} \qquad fields(\Sigma[\ell]) = \overline{T}\ \overline{f} \implies (S[\ell, i] = \ell'') \wedge (\Sigma[\ell''] <: T_i) \quad \ldots}{\Sigma \vdash S}$$

Figure 12. Partial store typing rule from FDJ [4].

tObject might have multiple types. The analysis picks one of those types nondeterministically as the label for a RuntimeObject.

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.11).

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. 8, the root object is in domain `lent`. 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. 13).

A context Ω keeps track of the set of RuntimeEdges. The judgment for creating a RuntimeEdge takes the form:

$$\Theta; \textsf{RuntimeObject} \vdash \Omega \implies \Omega'$$

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. 13 for examples).

4.2 Illustrative Example

The reader may wish to work through the rewriting rules on the earlier `DataAccess` example. The OOG in Fig. 8 shows the root object, as well as formal domains to clarify the object and edge pulling operations. Fig. 13 shows selected AbstractGraph tuples and some applications of the rewriting rules. $\looparrowright$ denotes the next generated fact and $\looparrowright^*$ denotes the fact at the fixed point. We manually labeled some nodes and edges in Fig. 8 with the RuntimeObjects and RuntimeEdges generated in Fig. 13.

We manually added to Fig. 8 a thick edge labeled as "X" as an example of an imprecise edge that the rules do *not* generate. This illustrates how OOG edges are more precise than those obtained by superimposing field reference edges based on the associations in a class diagram. 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` <: `Number`.

4.3 Soundness

We want a RuntimeGraph to be a *sound* approximation of the true runtime object graph (ROG) for any program run. An OOG is a depth-limited projection that preserves a RuntimeGraph's soundness by adding summary edges.

Our static analysis manipulates tuples extracted from the abstract syntax tree of a program with ownership domain annotations. By the soundness of the underlying type system, the store typing characterizes any execution of a well-typed program, and thus, any of its runtime object graphs.

Informally, we relate a ROG to the RuntimeGraph as follows:

- **Object and Domain Soundness:** Each object ℓ in a ROG has exactly one representative RuntimeObject in the RuntimeGraph. Similarly, each domain in the ROG is represented by exactly one domain in the RuntimeGraph. Furthermore, this mapping is consistent with respect to the ownership relation. If object ℓ is in domain d in the ROG, then the representative of ℓ is in the representative of d 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:** Edges in a runtime graph soundly abstract all field points-to relations between objects in an ROG. Namely, if there is a field reference from object ℓ_1 to object ℓ_2 in a 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.

We build on the formalization of ownership domains using Featherweight Domain Java (FDJ) [4]. We reproduce here parts of the store typing rule *T-Store* (Fig. 12). An overbar represents a sequence. In FDJ, locations represent object identity. A type $C{<}\overline{d}{>}$ consists of the class of an object and actual ownership domain parameters. By Rule *Aux-Owner*, the first actual domain, d_1, is the owner. A store S maps a location ℓ to its contents: the type of the object, and the values stored in its fields. $S[\ell]$ denotes the store entry for ℓ. $S[\ell, i]$ denotes the value in the ith field of $S[\ell]$.

Given an object in a ROG 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. Also, $owner(C{<}\overline{\ell'.n}{>}) = \ell'_1.n_1$. *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. $dom()$ returns the mathematical domain of a mapping. $domains()$ returns the ownership domains that a type declares [4].

Theorem: Object and Domain Soundness. Given a RuntimeGraph(Θ, Ω), H maps locations in the store to RuntimeObjects:

$\forall \Sigma \vdash S, \forall \Theta, \exists H : dom(S) \mapsto dom(\Theta), \forall \ell \in S$

$\Sigma[\ell] = C{<}\overline{\ell'.n}{>}$ and $owner(S[\ell]) = \ell'.d$

$\implies H[\ell] = (\{o_1 \ldots o_n\}, d)$ and

$d' \in domains(C) \implies \exists i \in 1..n$. typeof$(o_i) = T_i$ and $d' \in domains(T_i)$

$(d_{lent}, NULL)$ AbstractDomain for built-in domain `lent` – NULL parent (1)

$(t_{UnitTest}) \quad (t_{DataAccess}) \quad (t_{ArrayList<Integer>})$ AbstractTypes for the type declarations (2)

$(o_{test}, d_{lent}, t_{UnitTest}, \{\})$ AbstractObject for `test` (3)

$(t_{UnitTest}, [d_{DATA}], t_{DataAccess})$ AbstractEdge for `UnitTest`'s field reference to `dataAccess` (4)

$(d_{DATA}, t_{UnitTest}) \quad (d_{ENV}, t_{UnitTest})$ AbstractDomains for `DATA` and `ENV` (5)

$(o_{dataAccess}, d_{DATA}, t_{DataAccess}, \{d_{PENV} \mapsto d_{ENV}\})$ AbstractObject for `dataAccess` (6)

$(t_{DataAccess}, [d_{PENV}], t_{ArrayList<Integer>})$ AbstractEdge for field reference in `DataAccess` to `v2` (7)

$(t_{DataAccess}, [d_{STATE}], t_{Integer})$ AbstractEdge for field reference in `DataAccess` to `int1` (8)

$(t_{DataAccess}, [d_{STATE}], t_{Number})$ AbstractEdge for field reference in `DataAccess` to `num1` (9)

$(d_{STATE}, t_{DataAccess}) \quad (d_{PENV}, t_{DataAccess})$ AbstractDomains `STATE` and `PENV` (10)

$(o_{num1}, d_{STATE}, t_{Number}, \{\}) \quad (o_{int1}, d_{STATE}, t_{Integer}, \{\})$ AbstractObjects for `num1` and `int1` (11)

$(o_{v2}, d_{PENV}, t_{ArrayList<Integer>}, \{d_{ELTS} \mapsto d_{PENV}\})$ AbstractObject for field declaration `v2` (12)

$(t_{ArrayList<Integer>}, [d_{ELTS}], t_{Integer})$ AbstractEdge to `ArrayList`'s element from virtual field (13)

$(d_{ELTS}, t_{ArrayList<Integer>})$ AbstractDomain to store `ArrayList`'s elements (14)

$(o_{obj}, d_{ELTS}, t_{Integer}, \{\})$ AbstractObject from virtual field on `ArrayList` (15)

R-CONVERT-OBJECT$(o_{test}) \looparrowright$ R-NEW-OBJECT$(o_{test}) \looparrowright$ RuntimeObject$(\{o_{test}\}, d_{lent})$ (vo1)

R-CONVERT-OBJECT$(o_{num1}) \looparrowright$ R-NEW-OBJECT$(o_{num1}) \looparrowright$ RuntimeObject$(\{o_{num1}\}, d_{STATE})$ (vo2)

R-CONVERT-OBJECT$(o_{int1}) \looparrowright$ RuntimeObject$(\{o_{num1}\}, d_{STATE}) \in \Theta$ and $o_{num1} : Number$
and $o_{int1} : Integer$ and $Integer <: Number \looparrowright$ R-MERGE-OBJECTS(o_{int1})
$\looparrowright$ replace RuntimeObject$(\{o_{num1}\}, d_{STATE})$ with RuntimeObject$(\{o_{num1}, o_{int1}\}, d_{STATE})$ (vo3)

R-CONVERT-OBJECT$(o_{dataAccess}) \looparrowright$ 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$(o_{dataAccess}, d_{PENV}, d_{ENV}) \looparrowright$ try$(\{o_{v2}\}, d_{ENV}) \looparrowright^{*}$ RuntimeObject$(\{o_{v2}\}, d_{ENV})$ (vo5)

$(\{o_{dataAccess}\}, d_{DATA})$ and AbstractEdge(7) $\looparrowright$ RuntimeEdge$([d_{DATA}], t_{DataAccess}, [d_{DATA}, d_{PENV}], t_{ArrayList<Integer>})$ (ve1)

Binding $d_{PENV} \mapsto d_{ENV}$ on (6) and R-PATH-SUBST maps $[d_{DATA}, d_{PENV}]$ to $[d_{ENV}]$ (newpath1)
$\looparrowright^{*}$ RuntimeEdge$([d_{DATA}], t_{DataAccess}, [d_{ENV}], t_{ArrayList<Integer>})$ (ve1.1)

$(\{o_{v2}\}, d_{PENV})$ and AbstractEdge on (13) $\looparrowright$ RuntimeEdge$([d_{PENV}], t_{ArrayList<Integer>}, [d_{PENV}, d_{ELTS}], Integer)$ (ve2)

Binding $d_{ELTS} \mapsto d_{PENV}$ on (12) and R-PATH-SUBST maps $[d_{PENV}, d_{ELTS}]$ to $[d_{PENV}]$ (newpath2)
$\looparrowright^{*}$ RuntimeEdge$([d_{PENV}], t_{ArrayList<Integer>}, [d_{PENV}], t_{Integer})$ (ve2.1)

Figure 13. Rewriting rules illustrated on the `DataAccess` example.

Definition: Edge Soundness. Given a Σ, S, a RuntimeGraph(Θ, Ω) and H that fulfill the above *object and domain soundness*, we state *edge soundness* as follows (we do not yet have a proof of this):

$$\begin{aligned}
&\forall \Sigma \vdash S, \forall \ell_1, \ell_2 \in S \\
&S[\ell_1, i] = \ell_2 \implies H[\ell_1] = (\{o_1 : T_1 \ldots\}, d_1) \\
&\qquad\qquad \text{and } H[\ell_2] = (\{o_2 : T_2 \ldots\}, d_2) \\
&\quad \text{and } \exists \text{RuntimeEdge}([\ldots, d_1], T_1', [\ldots, d_2], T_2') \in \Omega. \\
&(T_1' <: T_1 \text{ or } T_1 <: T_1') \text{ and } (T_2' <: T_2 \text{ or } T_2 <: T_2')
\end{aligned}$$

Proof of Object and Domain Soundness. The proof is by induction over the ownership tree. The owner of an object is set at creation time as an existing domain on an existing object, so the ownership relation is well-founded and has no cycles. The base case for the induction is trivial. The top-level object in the ROG has a unique representative in the RuntimeGraph corresponding to the root RuntimeObject. We strengthen the inductive hypothesis (i.h.) as follows: *In a ROG, each object of runtime type C is represented by exactly one RuntimeObject θ that merges an AbstractObject o of type C in the RuntimeGraph.*

The details are in the companion report [3]. The proof crucially relies on the unification of inherited domains in the AbstractGraph (Fig. 14). The proof requires several lemmas which are well-formedness rules on the RuntimeGraph. For example:

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.

Limitations. The proof assumes that objects are created only in locally declared domains or domain parameters and does not reflect the existence of `lent` or `unique` [4]. Indeed, an 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. This flow analysis is not currently implemented, so a `unique` object obtained from a factory method must be annotated with the domain to which it belongs. Similarly, a flow analysis can determine what domain a `lent` object is really in. Our implementation does not currently display objects that are annotated with `lent`, except for the root object. Unless the user requests otherwise, we purposely exclude objects that are `shared` since they often add uninteresting clutter (the analysis may also merge objects in the `shared` domain excessively).

5. Abstraction by Types

In addition to abstraction by ownership hierarchy, an OOG can provide abstraction by types. We motivate these features using a real object-oriented system, JHotDraw (www.jhotdraw.org).

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

In JHotDraw, 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`.

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. The analysis that we described until this point, produces for JHotDraw an OOG that merges too many architecturally relevant objects (see the outcome in the report [3]).

A key insight, however, is that there are no object creations of interface types. So to gain some precision, we can construct the AbstractGraph differently (Fig. 14). Line (c) generates a declaration-based view (DBV) by generating abstract objects for all field and variable declarations. In contrast, Line (d) considers only object creation expressions and generates an instantiation-based view (IBV), and is similar to how Rapid Type Analysis (RTA) determines a method call's receiver during call graph construction [6].

Using an IBV, the analysis never generates an abstract object of type `ViewChangeListener`. Rather, it creates abstract objects for types `SelectionTool` and `AlignCommand`. Then, the runtime graph keeps `AlignCommand` and `SelectionTool` distinct, since there is no subtyping relation between them. Thus, an IBV can keep `Command` and `Tool` distinct (`SelectionTool` $<:$ `Tool`, `ViewChangeListener` and `AlignCommand` $<:$ `Command`, `ViewChangeListener`).

Special Cases. Even in an IBV, the analysis must still handle variable declarations of interface types. For example, in JHotDraw, a `CommandMenu` object in domain `VIEW` declares a `Vector<Command>`, and places the `Vector`'s `Command` objects in a domain parameter `C`. Also, a virtual field indicates the presence of a `Command` abstract object in `Vector`'s `ELTS` formal domain that stores the elements. Recall that `Command` is an interface. So the analysis cannot pick a more precise type for that abstract object, nor can it ignore it and

```
1. For each type declaration $C$ in the program
   (a) Create AbstractType $t$
   (b) For each actual or formal domain in $C$
       i. Create corresponding AbstractDomain $d$
   (c) For each declaration $d\ C'<\overline{a}>\ o$ in $C$ (DBV) or else
   (d) For each creation $new\ C'<\overline{a}>(\ldots)$ 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 $d$
       iii. Create AbstractObject $o$
       iv. Create bindings $\{b \ldots\}$ from formals $\overline{f}$ of AbstractType $t'$ to actuals $\overline{a}$ of $t$
       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 $t'$
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 14. Visitor to generate the AbstractGraph.

be sound, as this abstract object is pulled and carries binding information to generate the appropriate RuntimeEdges. For instance, the analysis pulls a `Command` abstract object from `ELTS` to a domain parameter `C`, and transitively to `CONTROLLER`. It also creates a RuntimeEdge from `CommandMenu` to any subclass of `Command` in `CONTROLLER`, such as `RedoCommand`. If the analysis were to add a `Command` abstract object to the `ELTS` domain, this would result in the same 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. Moreover, 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 the abstract type of an implicit anonymous class.

5.2 Abstraction by Trivial Types

For JHotDraw, an instantiation-based view, as discussed above, lacks abstraction because it shows objects for `RedoCommand`, and `NewViewCommand`, as well as objects for `ConnectionTool`, `CreationTool`, etc. What we want 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, respectively. So the analysis adds an edge from `CommandMenu` to `RedoCommand`, `NewViewCommand`, etc. Moreover, a `Command` wraps another `Command`. So the resulting graph is almost fully connected (See the outcome in the report [3]).

To improve abstraction and reduce clutter, we defined one heuristic to 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. 10).

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.

For JHotDraw, the default trivial types produce an OOG that still suffers from too much merging [3]. 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. Similarly, `ConnectionTool` and `CreationTool` have `Tool` as their non-trivial LUB. But the analysis does not merge `ConnectionTool` and `RedoCommand` because their LUB, `ViewChangeListener`, is a trivial type.

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. 15). 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. The user 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 types in the list of design intent types, $\hat{t}$ 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.

For example, 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 most to least architecturally relevant (`Drawing` appears before `Figure`). As a result, the analysis merges objects of type `StandardDrawing` and `BouncingDrawing` with objects of type `Drawing` into one object. It 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.

6. Discussion and Related Work

Evaluation. We evaluated the analysis on several extended examples of representative medium-sized programs to answer the following research question: *Does an OOG have a reasonable abstraction level or does it suffer from too much or too little merging?* The companion technical report shows various OOGs for the subject systems we studied, totalling 38 KLOC [3]. We also conducted an on-site field study where we analyzed a 30 KLOC module from a proprietary commercial system of over 250 KLOC. In 35 hours, we were able to add the annotations to the module and extract a top-level architecture for review by a developer [2].

In comparison, Rayside et al. reported that a static object graph analysis based on RTA produced unacceptable over-approximations for most non-trivial programs [22].

Performance. We have not yet studied the runtime complexity of the analysis. But the performance seems satisfactory. Computing the OOG in Fig. 15 takes less than a minute on a modest Intel Pentium 4 CPU 3GHz with 1.5 GB of RAM. This time includes building the program's abstract syntax tree to retrieve the annotations, building the abstract graph, converting it into a runtime graph, and then projecting it into an OOG. Using an instantiation-based view is faster than a declaration-based view because it results in fewer abstract objects that the analysis must manipulate.

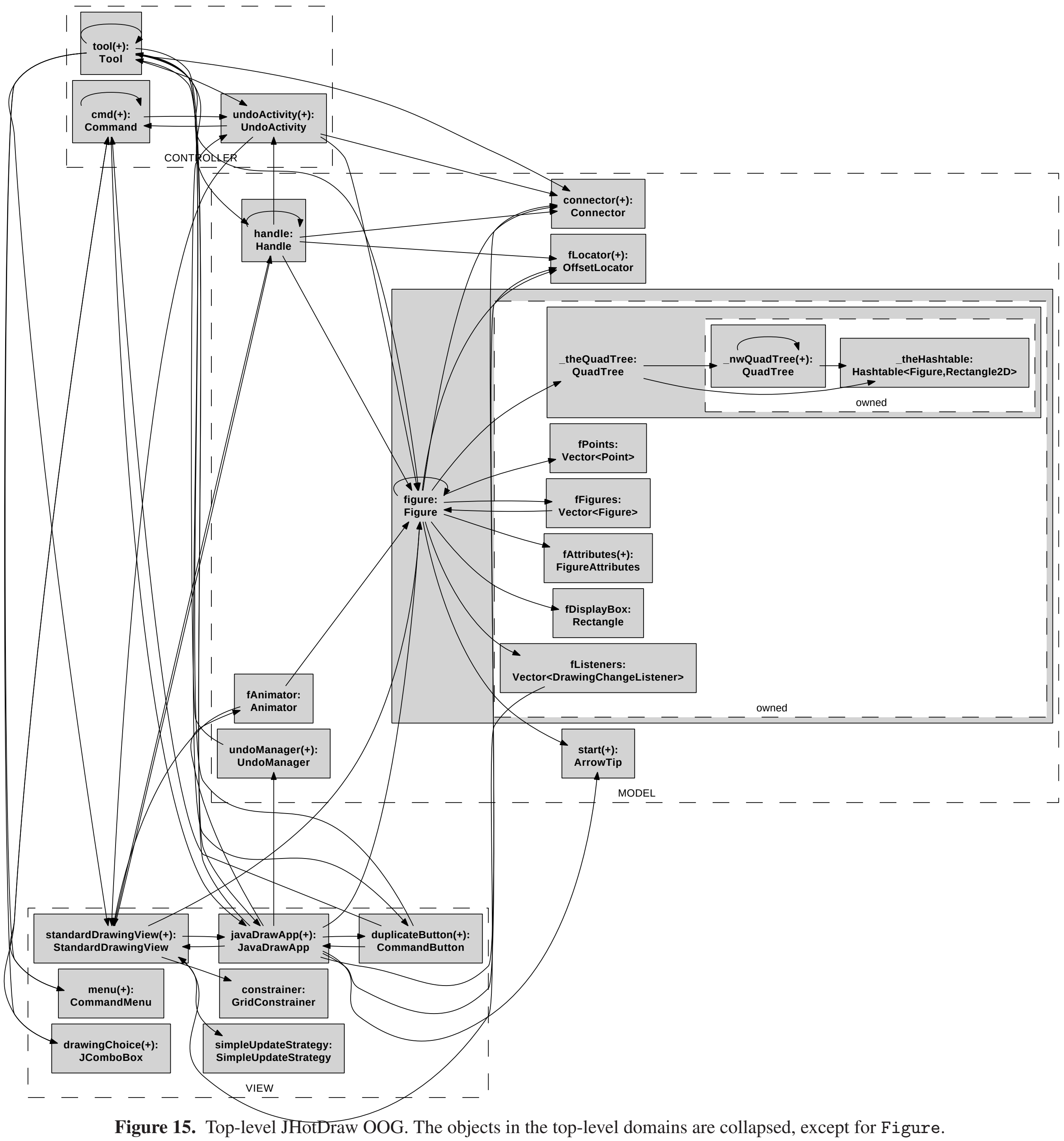
Architectural Extraction Process. Just as there are multiple architectural views of a system, there is no single right way to annotate a program. The best annotations produce a view comparable to what an architect might draw for an architecture. Good annotations minimize the number of objects in the top-level domains by pushing more objects underneath other objects.

A developer controls the architectural extraction process as follows. First, she chooses the top-level domains. Then, she achieves the desired number of objects in each top-level domain, through several strategies: (a) Pushing secondary objects underneath primary objects using strict encapsulation (private domains) or logical containment (public domains); (b) Passing low-level objects linearly whenever possible; and (c) Using abstraction by trivial types or design intent types to merge fewer or more objects. Finally, she achieves an appropriate level of visual detail by expanding or collapsing the substructure of selected objects, or changing the projection depth uniformly across all objects. The analysis adds any summary edges to account for the elided substructures.

Why Ownership Domains? The approach was presented in terms of the ownership domains type system, where each object contains 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 [8]. There is, however, a crucial expressiveness advantage in ownership domains that can reduce the number of objects in the top-level domains. In an owner-as-dominator type system, any access to a child object must go through its owning object [8]. In contrast, the ownership domains type system supports pushing almost any object underneath any other object in the ownership hierarchy. A child object may or may not be encapsulated by its parent object: a child object can still be referenced from outside its owner if it is part of a public domain of its parent, or if a domain parameter is linked to a private domain [4]. *Logical containment* with public domains is more flexible than the strict *encapsulation* of private domains, and helps reduce the number of objects in the top-level domains.

Architectural dynamism. An 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. Thus, our approach seems best suited for systems with little dynamic architectural reconfiguration.

Dynamic Analyses. There are several dynamic analyses for extracting or visualizing runtime structures [25, 10]. Static analysis, which considers all possible executions, is essential to extract sound information. DISCOTECT [25] recovers a non-hierarchical runtime architecture from a running system, one that shows one component for each instance created at runtime. DISCOTECT does



tool(+): Tool
cmd(+): Command
undoActivity(+): UndoActivity
CONTROLLER
connector(+): Connector
handle: Handle
fLocator(+): OffsetLocator
_theQuadTree: QuadTree
_nwQuadTree(+): QuadTree
_theHashtable: Hashtable<Figure,Rectangle2D>
owned
fPoints: Vector<Point>
figure: Figure
fFigures: Vector<Figure>
fAttributes(+): FigureAttributes
fDisplayBox: Rectangle
fListeners: Vector<DrawingChangeListener>
owned
fAnimator: Animator
undoManager(+): UndoManager
start(+): ArrowTip
MODEL
standardDrawingView(+): StandardDrawingView
javaDrawApp(+): JavaDrawApp
duplicateButton(+): CommandButton
menu(+): CommandMenu
constrainer: GridConstrainer
drawingChoice(+): JComboBox
simpleUpdateStrategy: SimpleUpdateStrategy
VIEW

Figure 15. Top-level JHotDraw OOG. The objects in the top-level domains are collapsed, except for `Figure`.

not require annotations, but instead, a developer maps low-level events from runtime traces to architectural elements.

More closely related are dynamic analyses that infer 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. produce matrix displays of the ownership structure [21]. Similarly, Mitchell 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 [17]. Noble, Potter, Potanin et al. showed both matrix and graph views of ownership structures and demonstrated that ownership is effective at organizing runtime object structures [11, 19]. We use the same key insight but in a static analysis that must address additional challenges.

Language Extensions. Specifying the architecture directly in code using language extensions simplifies the static extraction of runtime architectures [5, 24]. The ArchJava language specifies a architectural components directly in code, but prohibits returning references to instances of `component classes`, and this requires re-engineering existing implementations [5]. Ownership type annotations support common object-oriented idioms better, and allow returning references to objects. Thus, our annotation-based approach seems more adoptable for existing systems [2].

Object Graph Analyses. Several static analyses produce non-hierarchical object graphs without using annotations. PANGAEA [26] produces a flat object graph without an alias analysis and is unsound. WOMBLE [12] 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 [18] 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 [18, p. 248]. AJAX's heavyweight but precise alias analysis does not scale to large programs. In general, flat object 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.

Annotation-Based Systems. Lam and Rinard [14] 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 a fixed set of statically declared global tokens, and the result of the analysis is a graph showing which objects appear in which tokens. Using token parameters, the same code element can be mapped to different design elements depending on context. Unlike ownership domains, there is a statically fixed number of tokens, all of which are at the top level, so LR cannot show hierarchy, such as `listeners` nested within a `model` object (Fig. 2(b)). In contrast, declaring ownership domains within an object defines a sub-architecture of contained objects, and in the case of recursive types, the domain structure is hierarchical and unbounded in depth. The LR paper and formal system do not mention inheritance, and there is no proof of soundness of LR either with or without inheritance. LR's only case study was an order of magnitude smaller than JHotDraw (1.7KLOC). If we were to apply LR to JHotDraw anyway, ignoring inheritance, it would show at least 200 objects in the top-level tokens. In contrast, our system applies abstraction by ownership hierarchy and by types to show an order of magnitude fewer objects in the top-level domains.

A *package* in confined types [7], which track classes not instances, can be considered as a package-level static ownership domain, and thus, is coarser than an LR token.

Shape Analysis. Our analysis creates a graph that summarizes possible relationships among objects at runtime. Shape analysis, e.g., [23], 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. 2(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.

ACKNOWLEDGMENTS. This work was funded in part by NSF CAREER award CCF-0546550 and DARPA contract HR00110710019. The authors are very grateful to Robert J. Simmons, Neelakantan R. Krishnaswami and the anonymous reviewers for helpful comments.

References

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

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

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

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

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

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

[7] B. Bokowski and J. Vitek. Confined Types. In *OOPSLA*, 1999.

[8] D. Clarke, J. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

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

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

[11] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Visual Languages and Computing*, 13(3), 2002.

[12] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE Trans. on Softw. Eng.*, 27(2), 2001.

[13] R. Kazman and S. J. Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Automated Softw. Eng.*, 6(2), 1999.

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

[15] Y. Liu and A. Milanova. Ownership and Immutability Inference for UML-based Object Access Control. In *ICSE*, 2007.

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

[17] N. Mitchell. The Runtime Structure of Object Ownership. In *ECOOP*, 2006.

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

[19] A. Potanin, J. Noble, and R. Biddle. Checking Ownership and Confinement. *Concurrency and Computation: Practice and Experience*, 16(7), 2004.

[20] J. Potter, J. Noble, and D. Clarke. The Ins and Outs of Objects. In *Australian Softw. Eng. Conf.*, 1998.

[21] D. Rayside, L. Mendel, and D. Jackson. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *WODA*, 2006.

[22] D. Rayside, L. Mendel, R. Seater, and D. Jackson. An Analysis and Visualization for Revealing Object Sharing. In *Eclipse Technology eXchange (ETX)*, 2005.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. In *POPL*, pages 105–118, 1999.

[24] 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.

[25] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. *IEEE Trans. on Softw. Eng.*, 32(7), 2006.

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