A Static Analysis for Extracting Runtime Views from Annotated Object-Oriented Code

Marwan Abi-Antoun
Carnegie Mellon University,
5000 Forbes Avenue,
Pittsburgh, PA 15213
marwan.abi-antoun@cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University,
5000 Forbes Avenue,
Pittsburgh, PA 15213
jonathan.aldrich@cs.cmu.edu

Abstract

We demonstrate a static analysis for extracting instance-based hierarchical views showing the runtime object graph for object-oriented programs. The code is annotated with ownership domain annotations and with additional annotations to make the output more visually appealing.

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

General Terms Languages

Keywords AliasJava, ownership domains, object graphs

1. Introduction

Tools that automatically extract static code views (e.g., UML class diagrams) from source code have been available for a few years and are now integrated into various development environments. However, these module views showing the static organization of the source code do not explain the runtime structure of object-oriented programs where much of the functionality is determined by what object instances point to what other instances. For instance, in a module view, it is common to see several classes in the system depending on a single container class, whereas at runtime, different instantiations of such an element have distinct conceptual purposes and correspond to different design elements. Furthermore, the generated module views tend to lack hierarchical decomposition, with the hierarchy consisting mainly of namespace information.

2. Runtime Views

At runtime, an object-oriented implementation can be represented as an object graph in terms of nodes corresponding to objects and edges corresponding to communication (creation, usage, etc.) between objects. We are developing an analysis to statically approximate the runtime object graph of an annotated program. The key idea is to use ownership domain annotations to allow us to structure an object graph into an implicit ownership tree. In a type system with ownership domains, each object contains one or more domains and each object can be in exactly one domain [1]. Having the hierarchy is essential for the scalability of the approach since runtime object graphs can be large.

Different ownership domains and different places in the hierarchy give the analysis the ability to distinguish between instances of a class that would be merged in a class diagram allowing better understanding of the dynamic object structure of a system. In addition, domain names are specified by the developer and therefore can be used to communicate abstract design intent not just arbitrary aliasing information, so they ought to be more useful than the result of a static analysis without relying on annotations.

```
public class Main {
  domain data;
  domain userTier, logicTier, dataTier;

  link userTier -> logicTier, logicTier -> dataTier;

  private dataTier Data<data> objData;
  private logicTier Logic<dataTier, data> objLogic;
  private userTier Client<logicTier, data> objClient;
  ...
}
```

Figure 1. AliasJava code for a simple three-tiered course registration system.

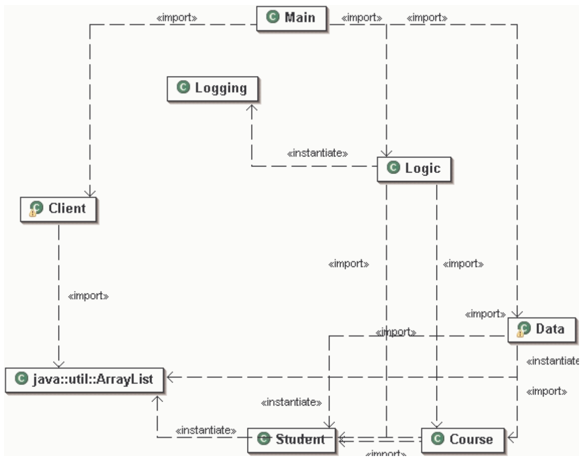

Figure 2. UML class diagram extracted from the original Java program using [3].

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

```
class C <owner, d>
    assume owner -> d {
domain a, b;
link b -> d;

Object< d > f;
Object< a > g;
Object< b > h;
}
```

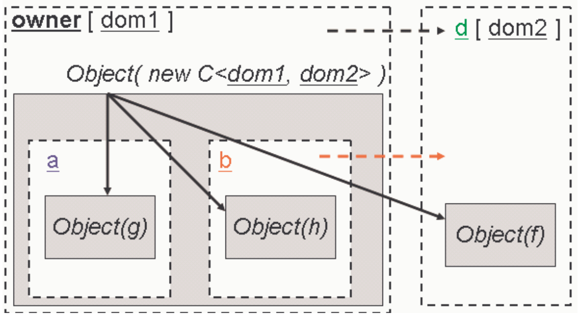

Figure 3. Domains a and b are declared in class C. Every object (gray) is in exactly one domain (dashed) where $Object < this.a > g$ means that $Object(g)$ is in domain a. Link declarations (dashed edges) specify that objects in domain b have permission to access objects in domain d. $Object(f)$ is pulled into domain d. Solid edges are object references.

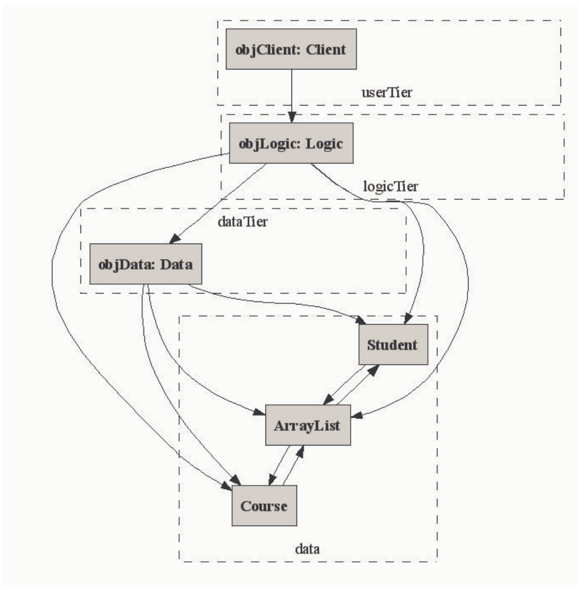

Figure 4. A runtime view extracted from the annotated source code in Figure 1. The `userTier`, `logicTier` and `dataTier` actual domains are shown as dashed rectangles. Objects `objUser`, `objLogic` and `objData` are shown inside their respective domains as gray boxes. `Student`, `Course` and `ArrayList` objects are in the actual domain `data`. Domain links are not shown in this diagram to reduce clutter.

3. Analysis

Our analysis relies on a set of annotations layered on top of ownership domain annotations. A static analysis uses the ownership domains annotations and the additional annotations to extract a runtime hierarchial instance view of a system.

The analysis distinguishes between *abstract* objects and domains which correspond to elements in the annotated program and *visualization* objects and domains which correspond to the visualization of the object graph. In the *abstract* object graph, domains and objects are declared in types, whereas in the *visualization* graph, we instantiate the types and show only the actual domains and object instances, i.e., we only show objects with their nested domains and objects inside of those domains. Edges between domains and edges between objects are also shown.

The analysis first extracts an abstract object graph from code with ownership annotations and then visualizes the abstract object graph. The visualization algorithm currently implements the following rules:

- **Merging**: In order to reduce the clutter in the views, we merge objects of the same type that are owned by the same domain.
- **Lifting**: The *visualization* object graph shows only actual domains and the objects inside them, i.e., for each formal domain, we determine the actual domain it is transitively bound to and we lift each object declared in formal domain transitively to show it in the actual domain (See Figure 3).

4. Example

Figure 1 shows the AliasJava code for a simple three-tiered course registration system. Figure 4 shows a runtime view extracted from the annotated source code in 1. A module view, such as the one shown in 2 may mislead a developer into thinking that instances of `java.util.ArrayList` are equally shared between objects of type `Logic`, `Data`, `Course`, and `Student`.

5. Implementation and Future Work

The current implementation consists of several visitor-based analyses using the AliasJava [2] infrastructure. The tool takes an AliasJava program and produces an output file for the visualization graph in the AT&T GraphViz dot format that can be converted to several image formats.

We are currently refining the analysis to produce a more precise output. We are also conducting case studies to evaluate the views we statically obtain for realistic object-oriented programs.

6. Related Work

Lam and Rinard proposed an approach using simple design time tags [3] which do not support hierarchy unlike runtime ownership domains. A dynamic analysis that does not rely on annotations is presented in [5].

References

[1] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, 2002.

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

[4] Omondo Eclipse UML. `http://www.omondo.com/`

[5] D. Rayside, L. Mendel and D. Jackson. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Fourth International Workshop on Dynamic Analysis*, 2006.