Tool Support for the Static Extraction of Sound Hierarchical Representations of Runtime Object Graphs

Marwan Abi-Antoun   Jonathan Aldrich

School of Computer Science, Carnegie Mellon University

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

Abstract

Ownership domain annotations specify in code architectural intent related to object encapsulation and communication. These annotations also enable the static extraction of a sound hierarchical representation of the runtime object graph.

The tool support consists of one Eclipse plugin to type-check the annotations inserted as Java 1.5 annotations, and another to extract a representation of the runtime object graph abstracted by ownership hierarchy and by types.

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

General Terms Experimentation, Languages

Keywords runtime architecture, architectural extraction

1. Introduction

Existing static analyses that extract representations of a system's instance structure produce low-level non-hierarchical object graphs, without annotations [4] or with annotations [5]. Such representations explain runtime interactions in detail but convey little architectural abstraction.

2. Approach Overview

We proposed an approach for extracting hierarchical representations of runtime object graphs, based on ownership annotations in the code [3]. The representation provides abstraction by ownership hierarchy and by types. It is also *sound* and accounts for all objects and relations that may exist at runtime [1].

Ownership Domains. For illustration, we use CourSys, a prototypical three-tiered course registration system. The first step is to add ownership domain annotations (Fig. 1).

```
1  interface IData {
2    public domain STATE;
3  }
4  interface ILogic<PSTATE> { // Domain parameter
5  }
6  class Data<PLOGIC> implements IData {
7    // Public domain gets unified with one on IData
8    public domain STATE;
9    STATE ArrayList<STATE Student> vStudent;
10   STATE ArrayList<STATE Course> vCourse;
11   PLOGIC ILogic<STATE> iLogic;
12 }
13 class Logic<PDATA, PSTATE> implements ILogic<PSTATE> {
14   private domain OWNED;
15   OWNED Logging log;
16   OWNED RWLock lock;
17   PDATA IData iData;
18 }
19 class Client<PLOGIC, PSTATE> {
20   PLOGIC ILogic<PSTATE> logicNode;
21 }
22 class Main {
23   domain USER, LOGIC, DATA; // Top-level domains
24   DATA final Data<LOGIC> objData;
25   LOGIC Logic<DATA, objData.STATE> objLogic;
26   USER Client<LOGIC, objData.STATE> objClient;
27 }
```

(a) CourSys with ownership domain annotations.

`DOM Type obj`: declare object `obj` of type `Type` in domain `DOM`;
[public] domain `DOM`: declare private [or public] domain `DOM`;
`class C<D_PARAM>`: declare domain parameter `D_PARAM` on `C`;
`C<DOM> cObj`: bind actual domain `DOM` to domain parameter;

(b) Simplified syntax for ownership domain annotations.

Figure 1. Annotated CourSys program.

The annotation system uses existing language support for annotations [2], but here we use language extensions.

Class `Main` declares three top-level ownership domains, `USER`, `LOGIC` and `DATA` (Line 23). The `Data` object instance is in the `DATA` domain, based on the annotation on the field declaration `objData` (Line 24). Next, class `Logic` declares a

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

private domain named `OWNED` and two objects, `log` and `lock` inside `OWNED` (Lines 14–16). The `Data` object has references to objects such as `Student` and `Course`. `Data` declares a *public domain* `STATE` (Line 8) to hold `Student` and `Course` objects and lists thereof.

A `Logic` object needs references to `Student` and `Course` objects that are in another domain. So class `Logic` declares a *domain parameter* `PSTATE` to access another object's state (Line 13). An instance of `Logic` gets access to the state objects when `Main` binds `Data`'s public domain to `Logic`'s domain parameter. An object's public domain is accessed using a syntax similar to a field access, and binding a formal domain to a domain parameter uses a syntax similar to Java generics (Line 25). A `Logic` object needs a reference to a `Data` object, so it declares another domain parameter, `PDATA` (Line 13), to get access to it. `Main` also binds `PDATA` on `Logic` to its `DATA` domain (Line 25).

Ownership Object Graph (OOG). A static analysis produces a hierarchical view of the system's runtime object graph, the Ownership Object Graph (OOG) [1]. In Fig. 2, the OOG shows object `objClient` in a `USER` tier, object `objLogic` in a `LOGIC` tier, object `objData` in a `DATA` tier. Objects of type `Student`, `Course` and lists thereof appear in a public domain of `objData`, `STATE`. Objects `log` and `lock` are inside `objLogic`'s `OWNED` domain.

Object Abstraction. Different executions may generate a different number of objects. The OOG summarizes multiple runtime objects with a canonical object. There are many `Student` objects at runtime, but only one in the diagram. Moreover, if two objects, *within the same domain*, might be aliased at runtime, they appear as a single object in that domain. Hence, the OOG does not show separate `iData:IData` and `objData:Data` objects in the `DATA` domain. The type system guarantees that two objects in different domains, such as `lock` and `_mutex`, can never be aliased.

Object Lifting. Domain parameters do not exist at runtime. The OOG shows all the objects that are in a given domain by transitively lifting objects from formal domains into actual domains. For instance, inside class `Logic`, object `iData:IData` is lifted from the formal domain `PDATA` into the actual domain `DATA`.

Hierarchy. The OOG folds lower-level objects into higher-level, more architecturally relevant objects. E.g., `objLogic` folds objects `lock` and `log`.

Edge Lifting. The OOG is depth-restricted. But child objects in a hierarchy may have fields that point to external objects, and these child objects may be beyond the visible depth. So the OOG adds summary edges from the parent objects to those external objects. For instance, the edge from `vRegistered:ArrayList<Course>` to `course:Course` summarizes the internal implementation that uses an array to holds references to the list elements.

Traceability. Each OOG element can be traced to a set of nodes from the program's abstract syntax tree.

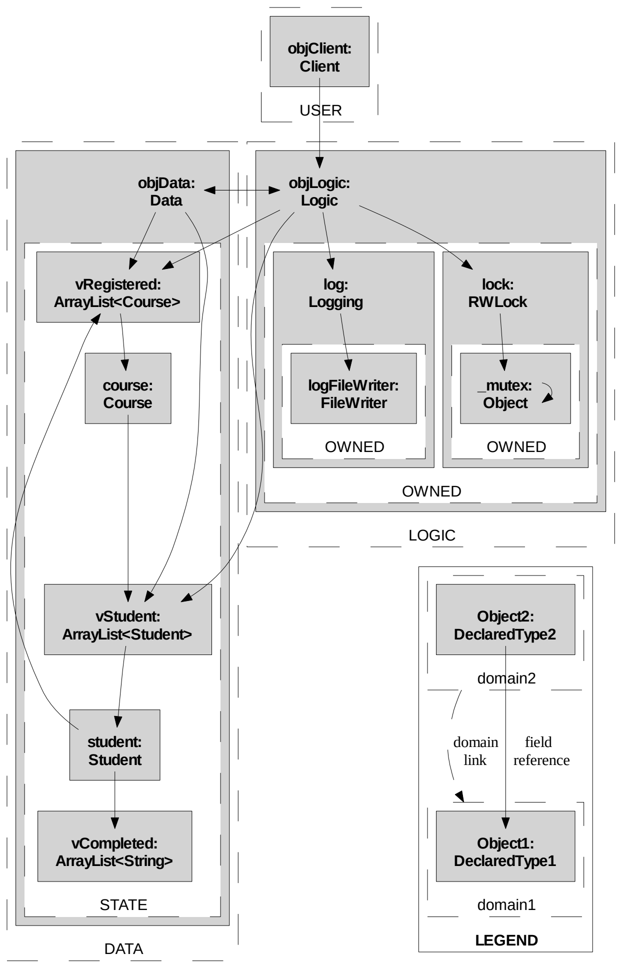

Figure 2. Ownership Object Graph for CourSys.

3. Conclusion

We presented tools to statically extract a sound hierarchical representation of the runtime object graphs of object-oriented programs with ownership domain annotations. The representation provides architectural abstraction by ownership hierarchy and by types.

References

[1] M. Abi-Antoun and J. Aldrich. Static Extraction of Object-Oriented Runtime Architectures. CMU-ISR-08-127, 2008.

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

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

[4] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *TSE*, 27(2), 2001.

[5] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, pages 275–302, 2003.