Static Extraction of Hierarchical Runtime Object Graphs – Tool Demonstration

Marwan Abi-Antoun    Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
Object-Oriented **Code vs. Runtime Structure**

“An object-oriented program's runtime structure often bears little resemblance to its code structure. The code structure [...] consists of classes in fixed inheritance relationships.

A program's runtime structure consists of [...] networks of communicating objects [...] Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.” (Gamma et al., 1994)
The following object diagram shows a snapshot of an application at run-time.

Object Diagram: a diagram of object structures which shows object instances exclusively.

Known Uses

[...]

Source: E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994. (CD-ROM edition)
Tool support to extract runtime structure less mature

- Low-level objects
- No architectural abstraction
- Some analyses incorrectly handle aliasing

JavaDrawApp, DrawingEditor represent one runtime object.

Output of Womble on JHotDraw (15 KLOC)
Ownership domain annotations enable the extraction of sound hierarchical object graphs using static analysis.
Extracting sound hierarchical object graphs using static analysis

• Why static analysis?
  • Dynamic analysis shows object graphs for a few program runs, not all

• Why sound?
  • To be most useful, show all objects and relations that could exist at runtime
Extracting sound hierarchical object graphs using static analysis

- Hierarchical graphs
  - Flat graphs do not provide architectural abstraction or scale
Demonstration Outline

- Ownership annotations
  - Adding annotations
  - Typechecking annotations
- Runtime structure
  - Extraction tool
- Real-World Example
  - JHotDraw
- Additional material
  - Static analysis
Ownership Domains
Ownership domains
[Aldrich and Chambers, ECOOP’04]

• Each object defines conceptual groups (ownership domains) to hold its state
• Separate object’s internals from object’s boundary (accessible to outside)
• Ensure private state not leaked
• Distinguish different “subsystems” within object
Example: Sequence

class Sequence {
    Cons head;

    public
    Iterator iterator() {
        return new Iterator(head);
    }
}

- Sequence has private state (head)
  - Should not be accessible to outside
- Sequence has iterators that are accessible to outside
  - Can also access private state
Sequence Code Structure

- `Sequence` is associated with `SequenceClient` via `~ seq`.
- `Cons` is associated with `SequenceClient` via `~ head` and `~ next`.
- `SequencelIterator` is associated with `Cons` via `~ current`.
- `Iter` represents the interface of `SequenceClient`.
Sequence: Private Domain

- Each object has one or more domains
  - E.g., Sequence declares domains owned and iters
- Each object is in exactly one domain
  - E.g., head in domain owned; iterator in domain iters

```java
@Domains("
  "owned"
)
class Sequence {
  @Domain("owned") Cons head;

  public Iterator iterator() {
    return new Iterator(head);
  }
}
```
Each object has one or more domains
- E.g., Sequence declares domains `owned` and `iters`

Each object is in exactly one domain
- E.g., head in domain `owned`; iterator in domain `iters`
Sequence Runtime Structure
Sequence Runtime Structure

- Collapse Sequence’s sub-structure
Encapsulation and Containment

(1) Strict encapsulation (private domain)

(2) Logical containment (public domain)
Annotation Tool Support

- Use **Java 1.5 annotations**
- Typechecker uses Eclipse JDT
- Warnings in Eclipse’s **problem window**
Demo: Checking Sequence

- Cannot return head of list
  - Head of list in **private domain**
  - **Stronger than making field private**
- Cannot nullify head of list
  - Stronger than Java visibility (e.g., **private**)
- Iterate over list
  - Iteror in **public domain**
Ownership Domain Parameters

@DomainParams({"elems"})
class Sequence {
    @Domain("owned<elems>")
    Cons head;
}

@DomainParams({"elems"})
class Cons {
    @Domain("elems") Object obj;
    @Domain("owner<elems>") Cons next;
}

- To share objects across domains
- Add domain parameter to hold elements in list
- **Implicit domain parameter “owner”** (Same as me, a.k.a. “peer” or “same”)
Demo: Annotating Listeners
(Iteration 1)
Listeners Example

- Listeners tricky in object-oriented code
- Reuse annotated Sequence
  - Disguised as ArrayList
Listeners Code Structure

```java
interface Listener {
}

class BaseChart
    implements Listener {
        List<Listener> listeners;
    }

class BarChart extends BaseChart {
}

class PieChart extends BaseChart {
}

class Model implements Listener {
    List<Listener> listeners;
}

class Main {
    Model model;
    BarChart barChart;
    PieChart pieChart;
}
```

Class diagram by Eclipse UML.
Demo: Listeners example

• Tool to add default annotations
  • Declare **owned** private domain
  • Private field place in domain **owned**
    • **owned**: object fully encapsulated
  • String mark **shared**
    • **shared**: shared persistently or globally
  • Method parameter mark **lent**
    • **lent**: temporary alias within method

• Not a smart inference tool!
Standard and third-party libraries

- Annotate external code
  - Ideally, library provider adds annotations
  - Annotations shared amongst authors
- Only annotate parts of library in use
- Wizard to generate skeleton XML file
Listeners Runtime Structure (version 1)

- Listeners at the top-level
Runtime Structure
Code Structure – Take 1

interface Listener {
}

class BaseChart implements Listener {
    List<Listener> listeners;
}
class BarChart extends BaseChart {
}
class PieChart extends BaseChart {
}
class Model implements Listener {
    List<Listener> listeners;
}
class Main {
    Model model;
    BarChart barChart;
    PieChart pieChart;
}
interface Listener {}

class BaseChart implements Listener {
    List<Listener> listeners;
}
class BarChart extends BaseChart {}
class PieChart extends BaseChart {}
class Model implements Listener {
    List<Listener> listeners;
}
class Main {
    Model model;
    BarChart barChart;
    PieChart pieChart;
}
Code vs. Runtime Structure

- Who points to who?
- Do not distinguish between conceptually different instances of same class
- Extra details: abstract classes, interfaces, etc.
- No hierarchy
Demo: Annotating Listeners (Iteration 2)
Change annotations

- Instance encapsulation
- May require changing code to avoid representation exposure, e.g.,
  - Return copy instead of alias to internal List
  - Pass object linearly
Listeners Runtime Structure (version 2)

model: Model

listeners: List<Listener>

barChart: BarChart

OWNED

listeners: List<Listener>

pieChart: PieChart

OWNED

OWNED

pieChart.OWNED != barChart.OWNED
Abstraction by Ownership Hierarchy

- Push **secondary** objects **under** primary objects using

(1) **Strict encapsulation** (private domain)

(2) **Logical containment** (public domain)
Hierarchy Provides Abstraction

- Can collapse object sub-structure
- Summary edges account for hidden objects
Tool Features

- Control projection depth
- Collapse/expand substructure
  - Selected domain or
  - Selected object
- Summary edges
- Elide private domains
- Control object labeling
Case Study: JHotDraw
Annotation/Extraction Process

1. Add
   - Annotations

2. Control Display
   - Extract As-Built Runtime Graph

3. Refine and iterate

4. (Optional) Change code and refine
   - Code
Annotation/Extraction Process

1. Choose top-level domains

2. Achieve desired number of objects in top-level domains
   a) Push secondary objects under primary objects
   b) Use abstraction by types to merge objects

3. Achieve appropriate visual detail
   a) Collapse or expand substructure of objects
   b) Change projection depth across all objects
JHotDraw: Code Structure

Manually generated UML Class Diagram for JHotDraw [Riehle, Thesis 2000].
JHotDraw: **Model-View-Controller (MVC)**
JHotDraw: Adding Annotations to Code

File: Main.java

class DrawApplication implements DrawingEditor ...
...
class MDI_DrawApplication extends DrawApplication ...
...
@DomainParams({"M", "V", "C"})
@DomainInherits({"MDI_DrawApplication<M,V,C>"})
class JavaDrawApp extends MDI_DrawApplication {
...
@Domains({"Model", "View", "Controller"})
class Main {
    @Domain("View<Model,View,Controller>")
    JavaDrawApp app = new JavaDrawApp();

    public static void main(
        @Domain("lent[shared]") String args[])
    {
        @Domain("lent") Main system = new Main();
    }
}
JHotDraw: “30-second Architecture”

- Hide contents of domains
  - Dotted edges summarize field references
  - Interestingly: no callback from M to C
JHotDraw: “30-minute Architecture”

Showing top-level domains and objects

LEGEND

RootObject: Type

Object1_with_Substructure (+): DeclaredType1

domain1

Object: DeclaredType

formal_domain

domain link

field reference

Object2: DeclaredType2

domain2

Object: DeclaredType

Object1_with_Substructure (+): DeclaredType1
JHotDraw: “30-minute Architecture”

Showing Drawing’s sub-structure

Output of Womble
Static Analysis
Static analysis

- Build **TypeGraph** from program’s AST
- Convert to **ObjectGraph** that soundly approximates all **runtime object graphs** (ROG)

**ROG**: graph where nodes represent runtime objects, edges represent reference or usage relations
TypeGraph: show types, domains inside types, and objects in domains

- Problem
- Approach
- Analysis
- Evaluation
- Conclusion
ObjectGraph: instantiate types, starting with root

- Problem
- Approach
- Analysis
- Evaluation
- Conclusion
ObjectGraph: instantiate types, starting with root
ObjectGraph: instantiate types, show domains and objects inside domains

- Problem
- Approach
- **Analysis**
- Evaluation
- Conclusion
ObjectGraph: instantiate types, show domains and objects inside domains

- Problem
- Approach
- Analysis
- Evaluation
- Conclusion
ObjectGraph: instantiate types, show domains and objects inside domains

- Problem
- Approach
- Analysis
- Evaluation
- Conclusion
ObjectGraph: instantiate types, show domains and objects inside domains

- Problem
- Approach
- Analysis
- Evaluation
- Conclusion
ObjectGraph: instantiate types, show domains and objects inside domains
ObjectGraph: instantiate types, show domains and objects inside domains
ObjectGraph: pull objects from formal domains to actual domains

- Problem
- Approach
- Analysis
- Evaluation
- Conclusion
**ObjectGraph**: pull objects from formal domains to actual domains
ObjectGraph: pull objects from formal domains to actual domains

- Problem
- Approach
- Analysis
- Evaluation
- Conclusion
ObjectGraph: merge objects, in one domain, that *may alias*, based on types

```java
class Model implements Listener {
    ...
}
```

● Problem ● Approach ● **Analysis** ● Evaluation ● Conclusion
ObjectGraph: add edges to represent field references

- Problem
- Approach
- **Analysis**
- Evaluation
- Conclusion
Conclusion

• Ownership domain annotations enable extraction of hierarchical runtime object graphs using static analysis
• Provide architectural abstraction by ownership hierarchy and by types