Assuring the Execution Architecture of Object-Oriented Programs using Ownership Domain Annotations

Marwan Abi-Antoun and Jonathan Aldrich
Carnegie Mellon University

Motivation: Why is the Execution Architecture Important?

Problem: a **software architecture** describes the structures of a system and the relationships among the elements, both within and between structures.
Assurance or **conformance checking** determines if a system is implemented according to its **as-designed architecture**. Otherwise, the validity of any **architectural analysis** will be suspect at best, and completely erroneous at worst.

The ***execution architecture*** shows the **runtime organization** of a system in terms of components and their interactions. We focus on **assuring** the execution architecture, since the latter is crucial for **performance**, **security**, **reliability**, distributed systems, etc.

Requirements on a Solution

To be adoptable, an approach to **extract** the **execution architecture** from **code** must:
- Work with **existing languages**, **frameworks**, **patterns** — i.e., **no language extensions**
- Deal with **aliasing**, **recursion**, **inheritance**, etc., in object-oriented programs

The extracted execution architecture must convey:

- **Abstraction:** group runtime objects into runtime ***components*** that are meaningful; and group related components into runtime ***tiers***. A tier is a **conceptual partitioning** of functionality;

- **Summarization:** represent **finitely** different number of objects generated during different executions, i.e., represent **multiple objects** at runtime with one object;

- **Aliasing:** if two variables might be aliased at runtime, show them as a single element in the diagram;

- **Soundness: represent all** objects and relations that may exist at runtime;

- **Scalability:** the diagram size should remain mostly constant as the program size increases arbitrarily.

Manual architecture documentation goes out-of-date quickly!

Some of these names do not exist anymore in the code, e.g., DrawWindow!

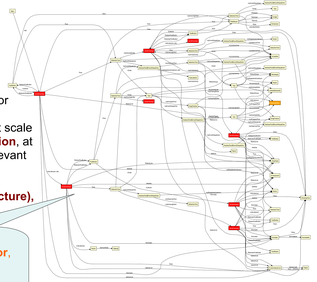
JHotDraw "Architecture Overview" in 'docs' folder in the source distribution [JHD].

Existing Approaches to Extract the Architecture

Dynamic Analyses:
- Results may not be repeatable
- **Static verification is ideal —** if it can be achieved

Static Analyses:
- Low-level object graphs useful for **detailed interactions**
- **Non-hierarchical** graphs do not scale
- Low-level objects, e.g., **Dimension**, at same level as architecturally relevant objects, e.g., **JavaDrawApp**
- Handle **aliasing** incorrectly
- **Cannot see** the **forest (architecture)**, for the **trees (objects)**

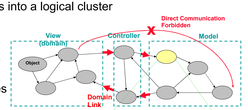
Multiple nodes, e.g., **JavaDrawApp**, **DrawingEditor**, — shown in **red** and **orange**, represent **one runtime object.**

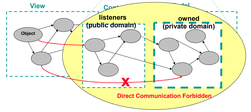


Thumbnail of the JHotDraw object graph, obtained by a static analysis tool, Womble [JW01], from the code without annotations.

What's an Ownership Domain Annotation?

- An ***ownership domain*** groups related objects into a logical cluster
  - Each object is in exactly one domain
  - An object contains one or more domains
- A domain name conveys **design intent**
- A domain can represent an **architectural tier**
  - E.g., "Model", "View", "Controller"
- References can only cross domain boundaries if there is a ***domain link*** between the two domains



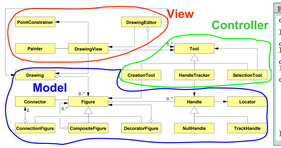


Implementation:
- Re-implemented as **Java 1.5 annotations**
- Does not require language extensions
- Use Eclipse instead of research infrastructure
- Enables reusing existing tools
- See additional details elsewhere [AA07a]

Simplified Syntax for Ownership Domains Annotations

a B g1:	declare object **g1** of type **B** in domain **a**
[public] domain a:	declare private [or public] domain
class C < *d* >:	declare formal domain parameter ***d*** on class **C**
C <owned> c:	provide actual domain for formal domain parameter on instance of **C**
link b -> d:	give domain **b** permission to access domain **d**

Example: Adding Ownership Domain Annotations to JHotDraw



Manually generated UML class diagram for JHotDraw, showing the core types (Source: [Rie02]).

```
File: Main.java
class DrawApplication <M, V, C> ...
    implements DrawingEditor <M, V, C> ... {
}
class MDI_DrawApplication <M, V, C> ... {
}
class JavaDrawApp <M, V, C> ...{
}
class Main {
  domain Model, View, Controller;
  View JavaDrawApp< Model,View,Controller> app = new ...;
}
```

Annotations on the top-level class in JHotDraw, using the simplified syntax. See concrete syntax in [AA07a].

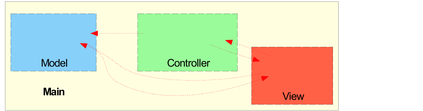
- **Annotate** JHotDraw according to **Model-View-Controller (MVC) design pattern**
- Specify three top-level domains: **Model**, **View**, **Controller**
- Propagate corresponding domain parameters: **M**, **V**, **C**
- Annotation process described in detail [AA07a]

- **Model:** includes instances of **Drawing**, **Figure**, **Handle**, etc.
  a **Drawing** is composed of **Figures** which know their containing **Drawing**
  a **Figure** has a list of **Handles** to allow user interactions
- **View:** includes instances of **DrawingEditor**, **DrawingView**, etc.
- **Controller:** includes instances of **Tool**, **Command**, and **Undoable**
  a **Tool** is used by a **DrawingView** to manipulate a **Drawing**
  a **Command** encapsulates an action to be executed

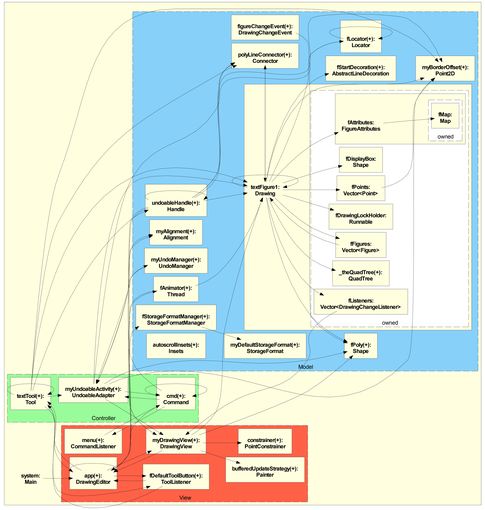
Why Ownership Domains?

- **Key insight:** ***apply ownership types to architectural extraction***
- Ownership types have other benefits, e.g., **encapsulation**, program verification, etc.
- Ownership domain annotations support **abstract reasoning** about **data sharing** by assigning each object at runtime to a single ownership domain
- A domain has an instance **granularity** larger than an object and conveys **design intent**
- Ownership adds **hierarchy** to the object graph and provides **precision about aliasing**
- Ownership domains are more **expressive** than strict *owner-as-dominator* type systems

Example: Extracting JHotDraw's Execution Architecture



The "30-second architecture": dotted edges summarize the field reference edges between objects inside those domains.



The "30-minute architecture": the edges represent field references. The symbol (+) is appended to an object's label when its substructure is elided. The graph was laid out automatically by GraphViz without user intervention.

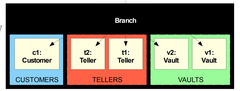
Some benefits of the extracted architecture:
- Provides overviews of the system architecture at **various levels of abstraction**
- Effectively **abstracts instances** compared to a raw object graph
- Effectively **abstracts related types**
- Helps **identify design problems** by relating the execution and the code architecture
- The **hierarchy** corresponds to the **system decomposition** in architectural diagrams
- Can be **extracted quickly**, then optionally and iteratively **refined**
- Conveys **object encapsulation**
- Shows **object communication**
- Suggests **structural constraints** that an architectural differencing tool could enforce

How to Extract an Architecture from an Annotated Program?

```
class Branch <CUSTOMERS> { /* Domain parameter */
  domain TELLERS, VAULTS;
  link TELLERS -> VAULTS;

  CUSTOMERS Customer c1; /* Use domain parameter */

  TELLERS Teller t1; /* Use public domain */
  TELLERS Teller t2;

  VAULTS Vault v1;
  VAULTS Vault v2;
}

class Bank {
  domain owned; /* Private default domain */

  /* Bind Branch <CUSTOMERS> formal to 'owned' actual */
  owned Branch<owned> b1;

}
```

Branch declares two domains,
- **TELLERS for Teller objects, and**
- **VAULTS for Vault objects.**

Branch also declares a domain link from TELLERS to VAULTS to allow Teller objects to access Vault objects.
Branch also takes a CUSTOMERS formal domain parameter that holds Customer objects.
The Bank object references a Branch object in field b1, binding the CUSTOMERS formal domain of Branch to the Bank's own private domain owned.



Step 1: A black-filled box represents a type, with color-filled domains declared inside it, and yellow objects declared inside each domain.

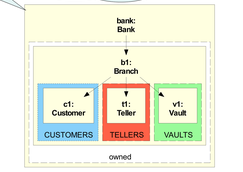
Challenge: How to show a hierarchy, i.e., show objects having children?
- Not suitable as an architecture
- Does not show objects with nested domains, and objects inside those domains

Challenge: How to summarize objects?
- Different executions can generate different number of objects
- Summarize multiple runtime objects with a canonical object at compile-time

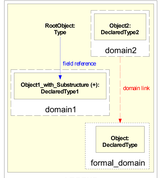
Challenge: How to deal with ownership domain parameters?
- The source code does not directly show what objects are in each domain
- Take each object declared inside a formal domain parameter, e.g., c1.
- **Pull object** into each domain that is bound to the formal domain, e.g., **CUSTOMERS** bound to **Bank.owned**.
- Do not show formal domain parameters

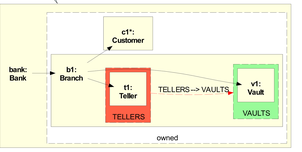
Challenge: How to deal with infinite graphs?
- Unroll graph to **limited depth**
- Add **summary edges** to parent objects when sub-objects point to external objects
(Not applicable in this example)

Could t1 and t2 be the same object at runtime?



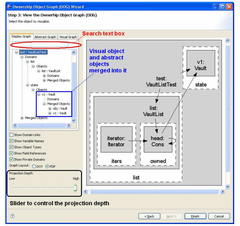
Step 2: Showing an instance of a Branch object:
- **Abstractly instantiate type Branch**
- **Merge objects of the same type that are in the same domain**
  - **merge objects t1 and t2 in domain TELLERS**
  - **merge objects v1 and v2 in domain VAULTS**
- **Object c1 is still in the CUSTOMERS formal domain parameter (shown with a dotted instead of a dashed border).**





Step 3:Pull object `c1`* from the formal domain parameter CUSTOMERS into the actual domain `Bank.owned` to which it is bound. Add a dashed edge to represent the domain link between TELLERS and VAULTS .

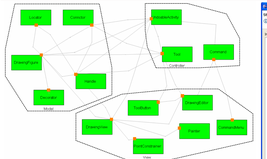
Tool Support for Extracting the Execution Architecture



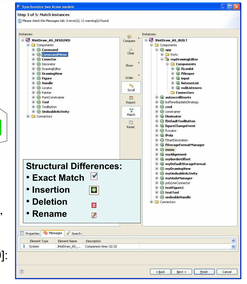
Screenshot of the tool to extract the architecture.

Tool Features:
- View **ownership tree** in left pane
- View **depth-limited projection** in right pane
- **Double click** on object in the diagram to select it in the ownership tree
- **Increase/decrease projection depth** using slider control
- **Track objects merged** into a visual object
- **Search by name** for object or domain
- **Elide private domains** with one checkbox
- **Hide/show domain links** with one checkbox
- **Hide/show the internals** of the selected object or domain with a right-click context menu
- **Label objects** using various options, as **obj: T**, where:
  - **obj** — optional instance name
  - **T** — optional type name

Tool Support for Assuring the Execution Architecture



As-designed architecture into a Component-and-Connector (C&C) View, represented in the Acme ADL. Model, View and Controller are represented as Acme groups. The as-designed architecture has one `DrawingFigure` component, instead of two components, `Drawing` and `Figure`, because in the code, `Drawing` *is-a* `Figure`.



Screenshot of the tool to compare architectural views. The tool recognizes all the top-level components correctly, including ones that have very different names, e.g., `DrawingEditor` (as-designed) vs. `app` (as-built).

1) **Map** to a general purpose Architecture Description Language (ADL), Acme [GMW00]:
- **Domain** maps to an Acme **group**
- **Hierarchy** maps to Acme **representation**
- **Predicates** enforce **structural constraints**

2) **Tool** for **differencing** and **merging hierarchical architectural** views [AAN+06]:
- Assume **no unique identifiers**
- Assume **disconnected/stateless operation**
- Assume **no exactly matching types**
- Assume **hierarchy** for **scalability**
- Detect **inserts and deletes**
- Detect **renames**
- Detect **hierarchical moves**
- Manually **force** or **prevent matches**

3) **Assurance**:
Compare **as-designed** architecture with the extracted **as-built** architecture.

For more information, please contact:
Marwan Abi-Antoun
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, U.S.A.
Email: **mabianto@cs.cmu.edu**

References

[AA07a] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *IWACO*, 2007.
[AA07b] M. Abi-Antoun and J. Aldrich. Compile-Time Views of Execution Structure Based on Ownership. In *IWACO*, 2007.
[AAN+06] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl, and D. Garlan. Differencing and Merging of Architectural Views. In *Automated Software Eng.*, 2006.
[AC04] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.
[JHD] JHotDraw. http://www.jhotdraw.org/, 1996.
[JW01] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE TSE*, 27(2):156—169, 2001.
[GMW00] Garlan, D., Monroe, R., Wile, D. Acme: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*, 2000.
[Rie02] D. Riehle. Framework Design: a Role Modeling Approach. PhD thesis, ETH Zurich, 2000.