Dynamic Architecture Extraction*

Cormac Flanagan[1] and Stephen N. Freund[2]

[1] Dept. of Computer Science, University of California at Santa Cruz, Santa Cruz, CA 95064
[2] Dept. of Computer Science, Williams College, Williamstown, MA 01267

Abstract. Object models capture key properties of object-oriented architectures, and they can highlight relationships between types, occurrences of sharing, and object encapsulation. We present a dynamic analysis to extract object models from legacy code bases. Our analysis reconstructs each intermediate heap from a log of object allocations and field writes, applies a sequence of abstraction-based operations to each heap, and combines the results into a single object model that conservatively approximates all observed heaps from the program's execution. The resulting object models reflect many interesting and useful architectural properties.

1 Introduction

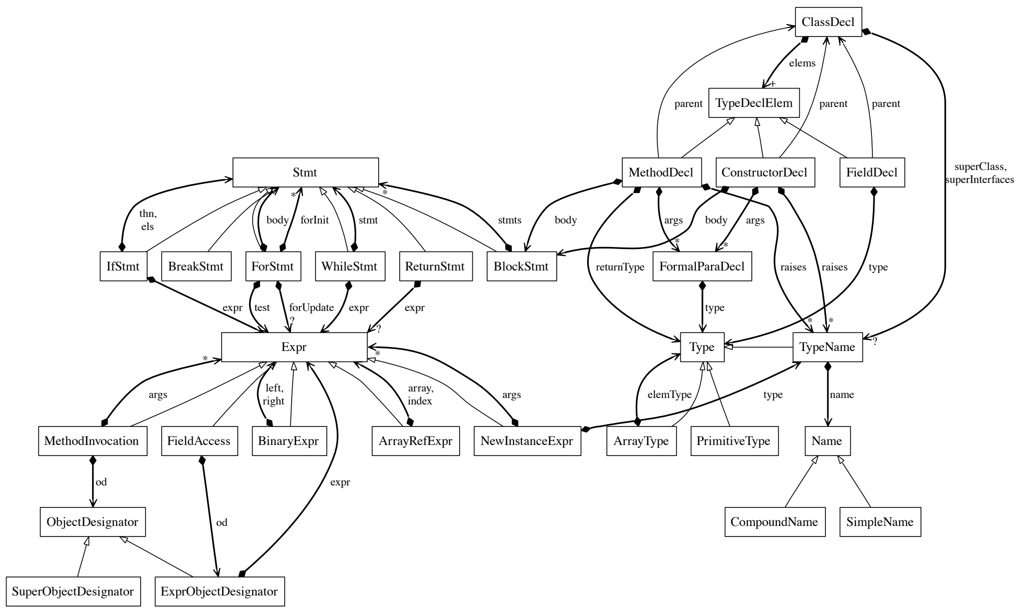
Object models capture the essence of object-oriented designs. However, many systems are developed without documented object models or evolve in ways that deviate from the original model. Tools to reconstruct object models are a valuable aid for understanding and reasoning about such systems. This paper presents a dynamic analysis to extract object models from existing code bases. We have found that these inferred object models explicate key structural invariants of object-oriented designs.

As an illustrative example, Figure 1 shows the inferred object model for parts of the abstract syntax tree (AST) data structure related to class declarations from the ESC/Java code base [15]. This object model is drawn as a UML class diagram [7], in which nodes represent classes and edges represent indicate both association and generalization relationships between classes. The graph reveals a number of important (and occasionally surprising) properties of ASTs:

- Each `ClassDecl` (at the top of the graph) has a `superClass` field with the somewhat unexpected multiplicity label '?', indicating that this field may be null. Inspection of the code revealed that the `superClass` field can in fact be null in one special case, namely when the `ClassDecl` is for `java.lang.Object`, the root of the class hierarchy.
- Each `ClassDecl` has a field `elems` containing one or more `TypeDeclElem` objects, as indicated by the multiplicity '+'. Again, this label was somewhat unexpected, since empty class declarations are valid in Java. However, further investigation revealed that the parser automatically adds an implicit nullary constructor to such classes.

* To appear at FATES/RV 2006.

Figure 1: Object Model for the AST Package from ESC/Java's Front-End

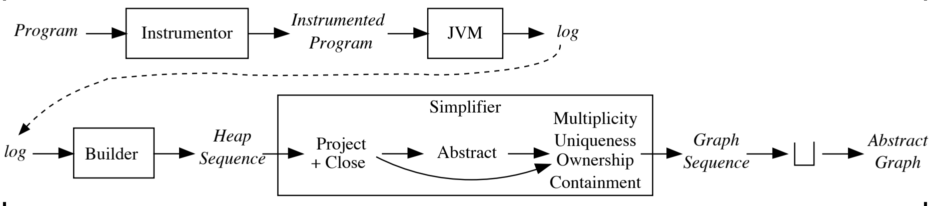


- Each `TypeDeclElem` may be a `MethodDecl`, `ConstructorDecl`, or `FieldDecl`, as indicated by the hollow-tipped generalization arrows from the subtypes to the supertype.
- Each `MethodDecl` contains zero or more `FormalParaDecl`s, as shown via the multiplicity '*'.
- The bold, diamond-tailed edges indicate unique references. All edges in this object model are unique, except for the `parent` pointers from each `TypeDeclElem`, which point back to the containing `ClassDecl`. Thus, the overall structure is mostly a tree, but documenting the non-unique `parent` pointers is crucial since any tree traversal algorithm must include special treatment for these pointers.
- Although not present in Figure 1, we also infer *ownership* and *containment* properties, which we found necessary to express encapsulation properties in complex situations where unique references are not sufficient, and we have enriched UML class diagrams to express these additional properties.

Object models could be reconstructed statically, by analyzing the program source code [5, 25, 23, 26]. However, precise static alias analysis is a notoriously difficult problem, and so static analyses have some difficulties inferring precise invariants regarding heap structure and sharing (although progress continues to be made on this topic).

In contrast, dynamic alias analysis reduces to a simple pointer comparison, and so dynamic analyses can provide very precise information regarding structural properties of heaps, such as: which portions of the heap follow a tree

Figure 2: Schematic



structure, which pointers are unique, and which objects are encapsulated within other objects. Of course, any dynamic analysis is limited by test coverage and may infer false invariants. In our experience, such anomalies, once discovered, are straightforward to rectify by appropriately extending the test inputs.

Figure 2 presents a schematic of our analysis tool, AARDVARK, which is based on *offline heap reconstruction*. It first executes an instrumented version of the target program that records a log of all object allocations and field writes. The *Builder* phase then uses this log to reconstruct a snapshot of the heap at each intermediate stage in the program's execution. The primary focus of this paper is on how to infer object models from these reconstructed heaps.

For each heap snapshot, AARDVARK isolates the relevant fragment of that heap via the *projection* and *closure* operations described in Section 2. It then uses *abstraction* (or object merging) to generate an initial object model for that heap, as described in Section 3. That object model is extended with additional information regarding multiplicities, unique pointers, ownership, and containment (see Section 4). Thus, the sequence of heap snapshots is abstracted into a corresponding sequence of object models.

We formalize the space of object models as labeled graphs, which form an abstract domain [12] with abstraction and concretization functions. Section 5 defines the upper bound operation $\sqcup$ on this domain, which we use to compute a single object model that conservatively approximates all of the heap snapshots from the program's execution.

The implementation of AARDVARK is described in Section 6. Preliminary experiments indicate that the inferred object models are quite precise and useful, and that they explicate important architectural details. In many cases, we can produce sufficiently accurate results by analyzing only a small sample of heap snapshots. Section 8 discusses some important topics for future work, including developing incremental versions of our abstraction algorithms.

2 Heap Projection and Closure

We begin by formalizing the notion of an object heap. We ignore primitive data (such as integers) and focus only on the structure of the heap. Let A be the set of object addresses (or simply objects) and let F be the set of field names in the program. We use $a, b, c, \ldots$ as meta-variables ranging over object addresses, and use f to range over field names. A *heap* H is a relation $H \subseteq A \times F \times A$

describing how fields of some objects point to other objects.[3] Each edge in H is written as $(a \rightarrow_f b)$, meaning that field f of object a points to object b.

In many situations, we may be interested only in certain parts of the heap, such as the objects corresponding to a particular package or data structure. If the object set $J \subseteq A$ describes these objects of interest, then the *projection* of a heap H onto J isolates them:

$$\mathit{proj}_J(H) = \{(a \rightarrow_f b) \;\mid\; (a \rightarrow_f b) \in H \;\wedge\; a, b \in J\}$$

Figure 3(a) shows a heap projection that focuses on the AST data structure of the ESC/Java front-end. The diagram shows that each class declaration contains a set of method, constructor, and field declarations.

This diagram also includes nodes that describe *how* class declarations are represented, via a `TypeDeclElemVec` object that contains an array. We often want to abstract away such low-level representation details, which is accomplished via the following *closure* operation that elides these intermediate objects (or *representation nodes*. For any set of low-level representation nodes $J \subseteq A$, the closure of a heap H with respect to J is defined by

$$\mathit{close}_J(H) = \left\{ (a \rightarrow_f b) \;\middle|\; \begin{array}{l} a, b \notin J \text{ and } \exists \text{ a path in } H \text{ from } a \text{ to } b \text{ whose first} \\ \text{field is } f \text{ and whose intermediate nodes are in } J \end{array} \right\}$$

The closure of Figure 3(a) with respect to representation nodes yields the diagram of Figure 3(b), which more directly shows the relationship between class declarations and their elements.

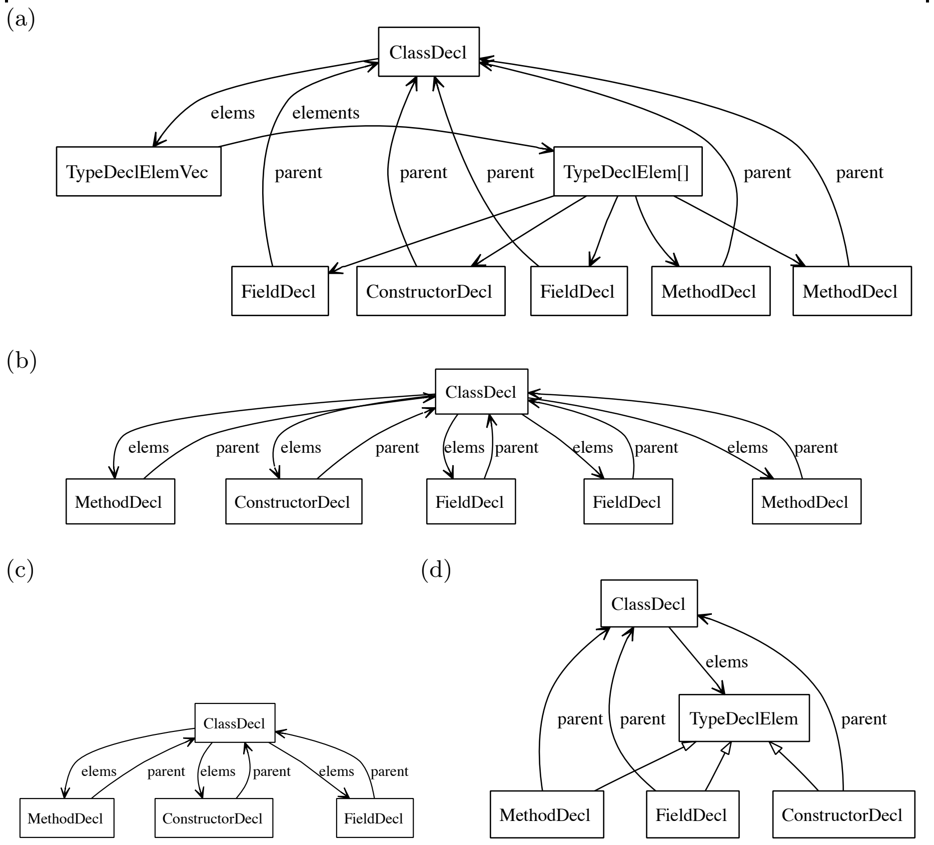
3 Abstraction

After projection and closure, the next step is to *abstract* from each program heap H (with perhaps millions of objects) a concise graphical representation G of the object model. Here, G is simply a graph over a collection of abstract nodes and edges, as defined precisely in Section 3.1. We consider a sequence of increasingly-precise abstractions. For clarity, we formalize the semantics of each representation with a *concretization function* γ that defines the meaning of a graph G as the set of heaps $\gamma(G)$ matching that description. Conversely, the *abstraction function* α maps a given concrete heap H to a corresponding graph $G = \alpha(H)$. For soundness, we require that the meaning of G includes the original graph H, *i.e.*, $H \in \gamma(\alpha(H))$.

A graph G_1 is *more precise* than G_2, denoted $G_1 \sqsubseteq G_2$, if $\gamma(G_1) \subseteq \gamma(G_2)$. Unlike in static analyses where the primary purpose of abstraction is to facilitate convergence, the purpose of abstraction in our setting is to ignore low-level details and isolate architectural invariants. For this reason, we do not require $\alpha(H)$ to be a most precise element of $\{G \mid H \in \gamma(G)\}$.

[3] We formalize the heap as a relation instead of a partial function $A \times F \rightarrow_p A$ to facilitate our subsequent development.

Figure 3: Closure, Abstraction, and Generalization



3.1 Heap Abstraction

The first class of abstractions we consider simply merges concrete objects into summary, or *abstract*, objects. Each abstract object $\hat{a}$ is a set of corresponding concrete objects (*i.e.*, $\hat{a} \subseteq A$), and we use $\hat{A}$ to denote the set of abstract objects. Thus $\hat{A} \subseteq 2^A$. An *(abstract) graph* G is a pair $(\hat{A}, \hat{E})$, where $\hat{E} \subseteq \hat{A} \times F \times \hat{A}$ is a set of field-labeled edges between abstract objects. Each abstract edge $(\hat{a} \rightarrow_f \hat{b})$ describes a set of possible concrete edges according to the *edge concretization function* γ:

$$\gamma(\hat{a} \rightarrow_f \hat{b}) = \{(a \rightarrow_f b) \mid a \in \hat{a}, b \in \hat{b}\}$$

Each abstract graph $(\hat{A}, \hat{E})$ represents a set of concrete heaps according to the concretization function γ^e:

$$\gamma^e(\hat{A}, \hat{E}) \quad = \quad \left\{H \;\middle|\; \forall e \in H.\ \exists \hat{e} \in \hat{E}.\ e \in \gamma(\hat{e}) \right\}$$

This function requires that every edge in the concrete heap H is represented by some corresponding edge in $\hat{E}$. (The superscript on γ^e distinguishes this concretization function from the ones presented below, and by convention the superscript always corresponds to the last component in the graph tuple.)

Applying this abstraction requires first determining a suitable collection of abstract objects. Since objects of the same class typically exhibit similar behavior, a particularly important abstraction is to partition objects according to their type, as shown in Figure 3(c). In this situation, $\hat{A}$ is a partition of A and is isomorphic to the set of non-abstract class types in the program. It is straightforward to define the corresponding abstraction function α^e:

$$\alpha^e(H) = (\hat{A}, \hat{E}) \quad \text{where } \hat{A} \text{ partitions } A \text{ according to type} \\ \text{and } \hat{E} = \left\{ (\hat{a} \rightarrow_f \hat{b}) \;\middle|\; \hat{a}, \hat{b} \in \hat{A} \wedge (\exists a \in \hat{a}, b \in \hat{b}.\ (a \rightarrow_f b) \in H) \right\}$$

Other possible partitioning strategies include, for example, partitioning objects according to their creation site, or merging objects that satisfy the same predicates, as in shape analyses [32]. Applying these ideas to object modeling remains for future work.

The following lemma states that α^e infers a conservative approximation of the given heap.

Lemma 1 (Soundness). $\forall H.\ H \in \gamma^e(\alpha^e(H))$.

We next extend our notion of abstraction to incorporate generalization (or subtype) edges, which are a key concern in object-oriented designs. Note that Figure 3(c) shows the `elems` field of a `ClassDecl` storing three different kinds of declarations. A better object model is shown in Figure 3(d), which indicates that `elems` stores a set of `TypeDeclElem`s, and the generalization edges (with hollow-tipped arrowheads) indicate that method, constructor, and field declarations are all subtypes of `TypeDeclElem`.

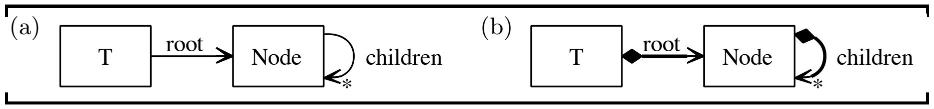
To illustrate how we perform generalization, suppose a class `A` extends `B`. We create corresponding abstract objects $\hat{a}$ and $\hat{b}$ as before, except that $\hat{b}$ now contains all concrete objects whose type is `B` or any subtype of `B`, including `A`. Thus $\hat{a} \subseteq \hat{b}$, and we indicate this containment relationship by drawing a generalization edge from $\hat{a}$ to $\hat{b}$.

The presence of generalization edges complicates the abstraction mapping. In general, a set of classes to which a field (such as `elems`) points could be generalized to any common supertype, but the best choice is the most-specific common supertype. Due to Java's multiple interface inheritance, this most-specific common supertype may not be unique, in which case AARDVARK employs simple heuristics to choose the most appropriate generalization.

4 Multiplicities and Structural Attributes

The abstractions of the previous section can produce precise summaries of large heaps, but they can also lose key information. This section enriches those abstract graphs with additional attributes describing the multiplicity of abstract edges, as well as sharing and structural properties of the heap.

Figure 4: Uniqueness



4.1 Multiplicities

The object model in Figure 1 labels each abstract edge with a multiplicity that describes how many objects it points to: "?" for at most one, "ϵ" for exactly one, "$*$" for zero or more, and "+" for one or more. Here ϵ indicates the absence of a multiplicity label. These multiplicity labels reveal that class declarations contain at least one element, method declarations contain zero or more formal parameters, and method declarations contain exactly one statement.

To compute this information, we annotate each abstract edge $(\hat{a} \rightarrow_f \hat{b})$ with a *multiplicity set* that describes, for each concrete object $a \in \hat{a}$, how many $\hat{b}$-objects the object a points to. Specifically, the multiplicity set $m((\hat{a} \rightarrow_f \hat{b}), H)$ of an abstract edge $(\hat{a} \rightarrow_f \hat{b})$ in a heap H is the set of natural numbers given by:

$$m((\hat{a} \rightarrow_f \hat{b}), H) = \{t_1, \ldots t_n\} \quad \text{where } \hat{a} = \{a_1, \ldots, a_n\}$$
$$\text{and } t_i = \Big| \{b \mid b \in \hat{b} \;\wedge\; (a_i \rightarrow_f b) \in H\} \Big|$$

We extend the abstract graph $(\hat{A}, \hat{E})$ with an additional component $M : \hat{E} \rightarrow 2^{\text{Nat}}$ describing the multiplicity sets of each abstract edge. The concretization function enforces this intended meaning, and the abstraction function computes the appropriate multiplicities from the concrete graph:

$$\gamma^m(\hat{A}, \hat{E}, M) = \Big\{ H \in \gamma^e(\hat{A}, \hat{E}) \;\Big|\; \forall \hat{e} \in \hat{E}.\ m(\hat{e}, H) \subseteq M(\hat{e}) \Big\}$$
$$\alpha^m(H) = (\hat{A}, \hat{E}, M) \;\text{ where } (\hat{A}, \hat{E}) = \alpha^e(H) \text{ and } M = \lambda \hat{e} \in \hat{E}.\ m(\hat{e}, H)$$

Lemma 2 (Soundness Of Multiplicities). $\forall H.\ H \in \gamma^m(\alpha^m(H))$.

Since multiplicity sets are rather dependent on the specific program execution, when drawing diagrams we generalize them to the more abstract multiplicity labels "?", "ϵ", "$*$", and "+", described above.

4.2 Uniqueness

The process of abstracting or object merging loses information about cycles or sharing in the underlying concrete heap. This limitation is illustrated by the abstract graph of Figure 4(a). From this graph, it is unclear whether the original heap was actually a binary tree, a doubly-linked list, a DAG, or some more general graph structure. To avoid this limitation, we next describe three increasingly sophisticated ways to enrich the abstract graph with additional information describing the degree to which sharing can occur in the underlying heap.

We begin by introducing the notion of unique edges. A concrete edge $(a \rightarrow_f b) \in H$ is *unique* if it points to an unshared object, that is, if H does not contain any other edge $(c \rightarrow_g b)$ that also points to b. This notion of uniqueness naturally extends to abstract edges: an abstract edge is unique if it only corresponds to unique concrete edges. In Figure 4(b), these unique edges (drawn in bold with a solid diamond on the tail) clarify that no sharing occurs, and thus this object model is more precise than Figure 4(a) since it describes only trees, and not other DAG or graph structures.

To formalize this notion of uniqueness, we extend the abstract graph $(\hat{A}, \hat{E}, M)$ of the previous section with an additional component $U \subseteq \hat{E}$ that describes which abstract edges are unique. The concretization and abstraction functions become:

$$\gamma^u(\hat{A}, \hat{E}, M, U) = \Big\{ H \in \gamma^m(\hat{A}, \hat{E}, M) \;\Big|\; \forall \hat{e} \in U.\ \forall e \in \gamma(\hat{e}) \cap H.\ e \text{ is unique in } H \Big\}$$

$$\begin{aligned}\alpha^u(H) = (\hat{A}, \hat{E}, M, U) \;\; & \text{where } (\hat{A}, \hat{E}, M) = \alpha^m(H) \\ & \text{and } U = \{\hat{e} \in \hat{E} \mid \forall e \in \gamma(\hat{e}) \cap H.\ e \text{ is unique in } H\}\end{aligned}$$

Lemma 3 (Soundness Of Uniqueness). $\forall H.\ H \in \gamma^u(\alpha^u(H))$.

4.3 Ownership

Unique pointers provide precise information in the ideal case where there is *no* sharing but cannot describe controlled or encapsulated sharing. For example, the concrete heap of Figure 5(a) includes two `java.util.LinkedList`s, each of which is represented by a doubly-linked list of `LinkedList$Entry`s. Each `LinkedList$Entry` contains a `Point`, except for the dummy node at the head of the list.

Even though `LinkedList$Entry`s are encapsulated by their owning list, pointers to `LinkedList$Entry`s are not unique. Thus, the abstract graph of Figure 5(b) loses this key encapsulation information and instead suggests that `LinkedList$Entry`s could be shared between `LinkedList`s.

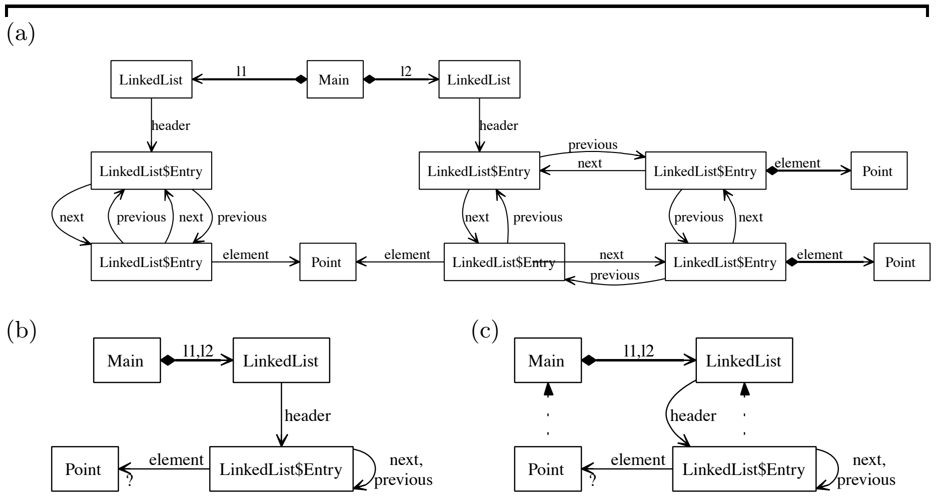
To remedy this limitation, we incorporate the notion of object *ownership* based on *dominators* [11]. An object a dominates object b if every path from a root of the heap to b must pass through a. Thus, the *dominates* relation $dom_H \subseteq A \times A$ for a heap H is the greatest fixpoint of the equations:

$$\begin{aligned} dom_H(b) &= \{b\} && \text{if } b \text{ is a root of } H \\ dom_H(b) &= \{b\} \cup \Bigg(\bigcap_{(a \rightarrow_f b) \in H} dom_H(a) \Bigg) && \text{otherwise} \end{aligned}$$

Roots could be, for example, all static fields in a program, or perhaps a more specific collection of objects, depending on the particular domain.

We extend this dominator relation to abstract graphs, and say that $\hat{a}$ *dominates* $\hat{b}$ *in* H (written $\hat{a} \triangleright_H \hat{b}$) if every $\hat{b}$-object is dominated by some $\hat{a}$-object, *i.e.*, if $\forall b \in \hat{b}.\ \exists a \in \hat{a}.\ a \in dom_H(b)$. When drawing abstract graphs, we indicate the closest, most precise dominator of each abstract object as an *ownership edge*, drawn as a dashed arrow. In Figure 5(c), these ownership edges show that

Figure 5: Ownership



each `LinkedList$Entry` object is owned by some `LinkedList`, which means that `LinkedList$Entry`s are never shared between `LinkedList`s. As expected, `Point`s are owned by the object `Main` and not the lists, since a `Point` is shared between both lists. For `LinkedList`, which is the target of a unique pointer, an ownership edge would be a redundant inverse of that unique pointer, and so is omitted.

We include this abstract domination relation $\triangleright_H \subseteq \hat{A} \times \hat{A}$ as an additional component in the abstract graph, whose concretization and abstraction functions become:

$$\gamma^{\triangleright}(\hat{A}, \hat{E}, M, U, \triangleright) = \left\{ H \in \gamma^u(\hat{A}, \hat{E}, M, U) \;\middle|\; \triangleright \subseteq \triangleright_H \right\}$$
$$\alpha^{\triangleright}(H) = (\hat{A}, \hat{E}, M, U, \triangleright_H) \;\text{ where } (\hat{A}, \hat{E}, M, U) = \alpha^u(H)$$

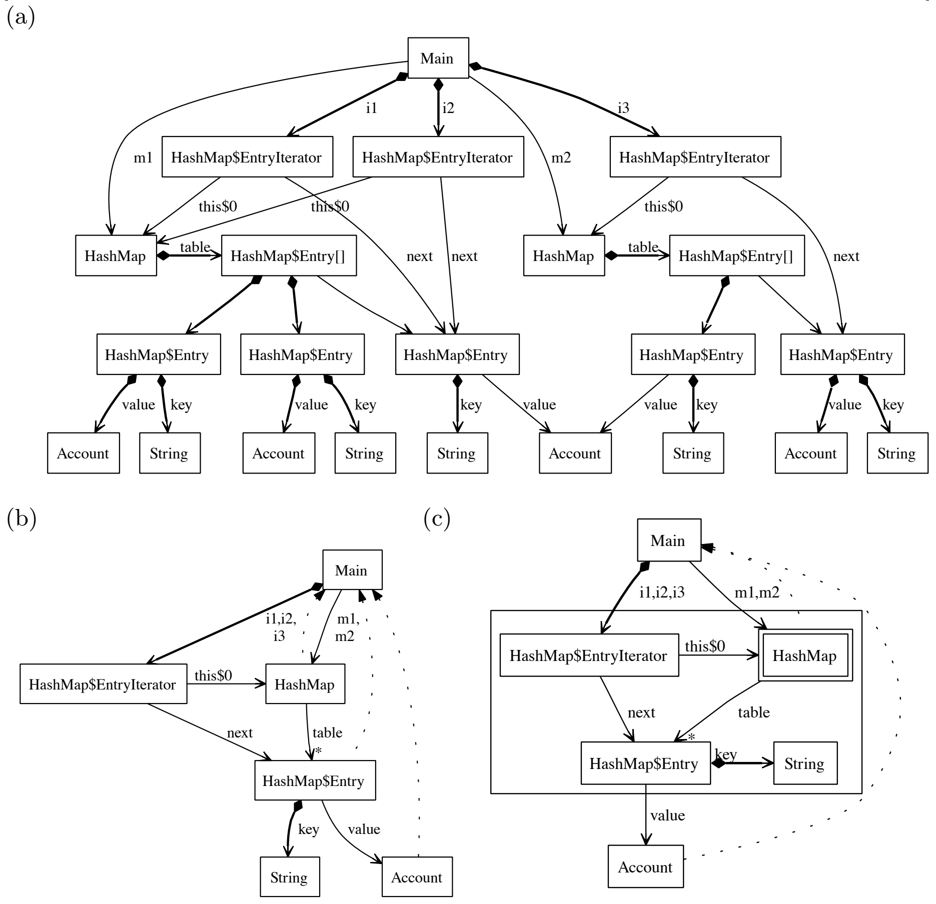
Lemma 4 (Soundness Of Ownership). $\forall H.\ H \in \gamma^{\triangleright}(\alpha^{\triangleright}(H))$.

4.4 Containment

Our final refinement captures encapsulation in complex situations for which neither uniqueness nor ownership suffices. Consider the concrete heap of Figure 6(a), which shows two `java.util.HashMap` objects, each of which has an array of `HashMap$Entry` objects and some iterators that also point to the `HashMap$Entry` objects. Each `HashMap$Entry` object is encapsulated in its `HashMap` and corresponding iterators. Thus, the `HashMap` representations can be partitioned into two connected components or *containers*.

However, in the abstract graph of Figure 6(b), neither uniqueness nor ownership is sufficient to explicate this partitioning. In particular, the only owner for the `HashMap$Entry`s is the object `Main`, since `HashMap$Entry`s are not dominated

Figure 6: Containment



by either HashMaps or the iterator. Thus, the graph suggests that HashMap$Entry objects could be shared between HashMaps, in which case updating one HashMap could then change the state of other HashMaps.

To remedy this limitation, we introduce the notion of *containment* and *singletons*. A container C is a set of abstract objects that represents an encapsulated data structure. For example, Figure 6(c) includes a container (drawn as a large box) encompassing the HashMap and related objects. Each concrete heap contains some number of *container instances* $c_1, \ldots, c_n$, where each container instance c_i is a set of concrete objects. The key concept of containment is that there can be no concrete edges between container instances.

Figure 6(c) also shows that HashMap is a *singleton* (drawn as a double box), indicating that each container instance contains exactly one HashMap object. Thus, each container instance includes a single HashMap and its associated iterators, entries, and `String`s, which means that `HashMap$Entry`s are never shared between `HashMap`s.

To formalize these notions, we extend abstract graphs with two additional components, the container C and the set of singleton objects S. The concretization function becomes:

$$\gamma^s(\hat{A}, \hat{E}, M, U, \rhd, C, S) \;=\; \left\{ H \in \gamma^{\rhd}(\hat{A}, \hat{E}, M, U, \rhd) \;\middle|\; \begin{array}{l} \exists n, c_1, \ldots, c_n. \\ \quad \forall \hat{a}, \hat{b} \in C, a \in \hat{a}, b \in \hat{b}.\ ((a \rightarrow_f b) \in H \Rightarrow \exists i.\ a, b \in c_i) \\ \wedge\ \forall \hat{a} \in S.\ \forall i \in 1..n.\ |\hat{a} \cap c_i| = 1 \end{array} \right\}$$

The corresponding abstraction function assumes we are given a fixed container C by the programmer[4]. This function infers the set of container instances by computing (using a union-find algorithm) the maximal partition P of the container objects into valid container instances, and it then computes the set S of singleton objects with respect to these containers:

$$\begin{array}{l} \alpha^s(H) = (\hat{A}, \hat{E}, M, U, \rhd, C, S) \\ \quad \text{where } (\hat{A}, \hat{E}, M, U, \rhd) = \alpha^{\rhd}(H) \\ \quad \text{and } P \text{ is the maximal partitioning of the container objects } \cup C \text{ such that} \\ \qquad \forall \hat{a}, \hat{b} \in C, a \in \hat{a}, b \in \hat{b}, \text{ if } (a \rightarrow_f b) \in H \text{ then } a \text{ and } b \text{ are in the same partition} \\ \quad \text{and } S = \{\hat{a} \in C \mid \forall c \in P.\ |\hat{a} \cap c| = 1\} \end{array}$$

Lemma 5 (Soundness Of Containment). *$\forall H.\ H \in \gamma^s(\alpha^s(H))$.*

5 From Heaps to Traces

The previous two sections show how to extract an abstract graph from each concrete heap. Applying this abstraction process to each observed heap $H_1, \ldots, H_n$ in the instrumented execution yields a sequence of graphs $G_1, \ldots, G_n$, where $G_i = \alpha(H_i)$. The final step is to merge these graphs into a single graph.

For this purpose, we introduce the following upper bound operation on abstract graphs $(\hat{A}, \hat{E}, M, U, \rhd, C, S)$. We assume that the graphs are defined over the same collection of abstract objects $\hat{A}$, heap roots, and container C. The upper bound operation then combines the remaining components by taking the union of the abstract edge sets; the point-wise union (denoted $\cup^m$) of the multiplicity maps; and the intersection of the unique edge sets, the domination relations, and the singleton sets:

$$\begin{array}{l} (\hat{A}, \hat{E}_1, M_1, U_1, \rhd_1, C, S_1) \ \sqcup\ (\hat{A}, \hat{E}_2, M_2, U_2, \rhd_2, C, S_2) \ = \\ \qquad\qquad (\hat{A}, \hat{E}_1 \cup \hat{E}_2, M_1 \cup^m M_2, U_1 \cap U_2, \rhd_1 \cap \rhd_2, C, S_1 \cap S_2) \end{array}$$

The point-wise union of two multiplicity maps is defined as:

$$(M_1 \cup^m M_2)(\hat{e}) = \begin{cases} M_1(\hat{e}) \cup M_2(\hat{e}) & \text{if } \hat{e} \in \mathit{domain}(M_1), \hat{e} \in \mathit{domain}(M_2) \\ M_1(\hat{e}) \cup \{0\} & \text{if } \hat{e} \in \mathit{domain}(M_1), \hat{e} \notin \mathit{domain}(M_2) \\ \{0\} \cup M_2(\hat{e}) & \text{if } \hat{e} \notin \mathit{domain}(M_1), \hat{e} \in \mathit{domain}(M_2) \end{cases}$$

[4] This technique generalizes to multiple different containers, and we are currently exploring ways to algorithmically or heuristically identify likely containers

Figure 7: Aardvark **Script for ESC/Java AST Package**

```
// g starts as the graph for the concrete heap.
// First, filter out nodes not reachable from ClassDecls:
g = proj(g, reachable(g, "ClassDecl"));
// Close over arrays and program-specific collections:
g = close(g, match(g, ".*\[\]|.*Vec|.*Set"));
g = abstractTypes(g);
g = generalizeTypes(g);
uniqueness(g); multiplicity(g);
ownership(g, match(g, "ClassDecl")); // ClassDecls are the roots
```

The next lemma states that operation $\sqcup$ is an upper bound operation on abstract graphs.

Lemma 6 (Upper Bound). *For any graphs G_1 and G_2:*

$$G_i \sqsubseteq G_1 \sqcup G_2 \qquad \textit{for all } i \in 1..2$$

We use this upper bound operation to combine the sequence of abstract graphs $G_1, \ldots, G_n$ into a single graphical summary $G = G_1 \sqcup \cdots \sqcup G_n$. The following lemma states that the final graph G is a conservative approximation for each observed heap in the program's execution.

Theorem 1 (Soundness for Traces). *Suppose $G_i = \alpha^s(H_i)$ for $i \in 1..n$ and that $G = G_1 \sqcup \cdots \sqcup G_n$. Then $H_i \in \gamma^s(G)$ for all $i \in 1..n$.*

6 Implementation

We have implemented our analysis in the Aardvark tool. Aardvark uses the BCEL binary instrumentor [6] to modify Java class files to record each object allocation and field write in a log file. The instrumentation overhead is roughly 10x–50x, depending on how memory-intensive the target program is. Currently, only single-threaded programs are supported, and analyzing concurrent programs remains for future work.

The off-line analysis then reconstructs a sequence of heaps from this log and applies the abstractions of the previous sections to each heap before finally merging the results into a single object model. The visual output is then generated by the `dot` utility [16].

A key characteristic of architectural diagrams is that they highlight concepts by eliding, or abstracting away, extraneous details, such as the representation nodes discussed in Section 2. Since which details are considered extraneous is domain-dependent, we intend our tool to be used in an interactive setting in which the software architect iteratively converges on an abstraction highlighting the desired architectural features. To support this methodology, Aardvark is extensible and driven by a script that configures and composes various predefined, or user-defined, abstractions. Figure 7 shows an example script.

Our prototype is capable of handling fairly large graphs. For example, the concrete heap used to construct Figure 1 contains 380,000 nodes and 435,000 edges. Aardvark reconstructs this concrete heap in 15 seconds and computes the abstract graph in another 15 seconds on a 3.06GHz Pentium Xeon workstation. While the concrete heaps for a trace are built incrementally, we currently do not use incremental abstraction algorithms, meaning that the tool cannot efficiently examine the several million intermediate heaps reconstructed from that log file. Instead, Aardvark samples these heaps in a configurable manner, which in most cases is sufficient to yield precise object models. Figure 1 was produced by sampling only the last heap from the log; all other graphs were produced by abstracting and merging all intermediate heaps, which required only a couple of seconds.

Further experimentation is needed to determine the best sampling technique for Aardvark, both in terms of performance and precision. Specifically, changing the granularity of logging from individual heap updates to, for example, method call boundaries, may lead to more precise models. The current low-level logging can reveal a method call's intermediate states in which structural invariants have been temporarily violated, resulting in imprecisions in the overall abstraction. We are currently designing incremental algorithms, which we expect to substantially improve scalability.

7 Related Work

Our tool produces graph representations similar in spirit, and based on, UML class diagrams [7]. Other tools extract some pieces of UML class diagrams from source code statically [5, 25, 23, 26], but they do not compute, or use unsound or imprecise heuristics to compute, the structural attributes we have discussed. Of these static tools, only SuperWomble [34] supports a limited form of user-defined abstraction. Ptidej [18] uses a dynamic analysis similar to Aardvark to refine statically-computed class diagrams. That tool does not explore the richer notions of ownership and containment or support user-defined abstractions.

Several studies have explored how to compute and visually present ownership information [21, 30, 28] from a program's heap. However, since no abstraction is performed, even small heaps can be too large to view effectively. More recently, Mitchell [27] shows how to compute ownership information for very large heaps in order to identify inefficiencies in a program's memory footprint. That approach uses a similar technique of repeatedly refining an initial heap configuration into an abstract summary, but it deals primarily with allocation and storage patterns and not other architectural issues. A number of other heap visualization tools also focus on memory or garbage collector profiling, *i.e.* [29, 22].

Several projects have dynamically inferred likely program invariants, including pre- and post- conditions [13], algebraic class specifications [20], and API usage requirements [35, 4]. Ernst *et al.* [14] have developed a technique to infer a class of common, but lower-level, data invariants for collection classes. We plan to generalize Aardvark to dynamically infer high-level, architectural specifications, such as usage patterns [19] and communication integrity constraints [1], which describe how components in a system may interact.

Shape analysis computes a set of abstract memory locations reachable from a pointer [24, 9, 17, 31, 32]. The goal of our work is similar to shape analysis in that we infer the relationships between objects in an abstract heap. One interesting avenue for future work is to extend Aardvark with additional notions of abstraction that compute shape information for use in subsequent dynamic or static analyses.

There are many static analyses for ownership and confinement, such as [11, 10, 3, 2, 8] and [33], respectively. For confinement, static analysis can ensure that specified containment relationships are not violated by leaking references to contained objects outside of a protection domain. While we capture similar containment relationships between objects, we have not focused on enforcing them. A dynamic enforcement mechanism may be an interesting, and perhaps more precise, alternative in some situations.

8 Conclusions and Future Directions

Tools for inferring architecture-level models can be very valuable for reasoning about legacy systems. This paper proposes that dynamic analysis is a promising approach that can identify not only relationships between types, but also interesting structural properties such as uniqueness, ownership, and containment. We see a number of interesting extensions and applications for this work, including:

- inferring (and enforcing) architecture-level specifications and invariants, including data dependent and temporal properties;
- seeding subsequent static, or dynamic, analyses with the shape information computed by our tool;
- exploring how object models evolve in large systems;
- inferring object models for lower-level languages such as C or C++; and
- supporting concurrency.

To support the studies of large systems, we are also currently developing incremental abstraction algorithms to improve scalability.

Acknowledgments: This work was supported in part by the National Science Foundation under Grants CCR-0341179 and CCR-0341387.

References

1. J. Aldrich. Using types to enforce architectural structure, 2006. Available at `http://www.cs.cmu.edu/~aldrich/papers/`.
2. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming*, pages 1–25, 2004.
3. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM Conference Object-Oriented Programming, Systems, Languages and Applications*, pages 311–330, 2002.
4. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *ACM Symposium on the Principles of Programming Languages*, pages 4–16, 2002.

5. ArgoUML. `http://argouml.tigris.org/`, 2006.
6. Byte Code Engineering Library. `http://jakarta.apache.org/bcel/`, 2006.
7. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide (2nd edition)*. Addison-Wesley, 2005.
8. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symposium on the Principles of Programming Languages*, pages 213–223, 2003.
9. D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *ACM Conference on Programming Language Design and Implementation*, pages 296–310, 1990.
10. D. G. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. In *European Conference on Object-Oriented Programming*, pages 53–76, 2001.
11. D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM Conference Object-Oriented Programming, Systems, Languages and Applications*, pages 48–64, 1998.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on the Principles of Programming Languages*, pages 238–252, 1977.
13. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.
14. M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington Department of Computer Science and Engineering, Seattle, WA, 1999.
15. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.
16. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice Experience*, 30(11):1203–1233, 2000.
17. R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symposium on the Principles of Programming Languages*, pages 1–15, 1996.
18. Y.-G. Guéhéneuc. A reverse engineering tool for precise class diagrams. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 28–41, 2004.
19. B. Hackett and A. Aiken. How is aliasing used in systems software?, 2006. Available at `http://glide.stanford.edu/saturn/`.
20. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *European Conference on Object-Oriented Programming*, pages 431–456, 2003.
21. T. Hill, J. Noble, and J. Potter. Scalable visualizations of object-oriented systems with ownership trees. *J. Vis. Lang. Comput.*, 13(3):319–339, 2002.
22. M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *MSP/ISMM*, pages 143–156, 2002.
23. D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *International Conference on Software Engineering*, pages 194–202, 1999.
24. N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symposium on the Principles of Programming Languages*, pages 66–74, 1982.
25. B. A. Malloy and J. F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *ACM Symposium on Software Visualization*, pages 105–114, 2005.
26. A. Milanova. Precise identification of composition relationships for UML class diagrams. In *Automated Software Engineering*, pages 76–85, 2005.
27. N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming*, 2006.
28. J. Noble. Visualising objects: Abstraction, encapsulation, aliasing, and ownership. In S. Diehl, editor, *Software Visualization: International Seminar*, pages 58–72, Dagstuhl, Germany, 2002. Springer.
29. T. Printezis and R. Jones. GCspy: An adaptable heap visualisation framework. In *ACM Conference Object-Oriented Programming, Systems, Languages and Applications*, pages 343–358, 2002.
30. D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *Workshop on Dynamic Analysis*, 2006.
31. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.
32. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symposium on the Principles of Programming Languages*, pages 105–118, 1999.
33. J. Vitek and B. Bokowski. Confined types in Java. *Software– Practice and Experience*, 31(6):507–532, 2001.
34. A. Waingold. Automatic extraction of abstract object models. Masters thesis, Department of Electrical Engineering and Computer Science, MIT, 2001.
35. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ACM International Symposium on Software Testing and Analysis*, pages 218–228, 2002.