Role-Based Exploration of Object-Oriented Programs

Brian Demsky
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
bdemsky@mit.edu

Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rinard@lcs.mit.edu

ABSTRACT

We present a new technique for helping developers understand heap properties of object-oriented programs and how the actions of the program affect these properties. Our dynamic analysis uses the aliasing properties of objects to synthesize a set of *roles*; each role represents an abstract object state intended to be of interest to the developer. We allow the developer to customize the analysis to explore the object states and behavior of the program at multiple different and potentially complementary levels of abstraction.

The analysis uses roles as the basis for three abstractions: role transition diagrams, which present the observed transitions between roles and the methods responsible for the transitions; role relationship diagrams, which present the observed referencing relationships between objects playing different roles; and enhanced method interfaces, which present the observed roles of method parameters.

Together, these abstractions provide useful information about important object and data structure properties and how the actions of the program affect these properties. We have used our implemented role analysis to explore the behavior of several Java programs. Our experience indicates that, when combined with a powerful graphical user interface, roles are a useful abstraction for helping developers explore and understand the behavior of object-oriented programs.

1. INTRODUCTION

This paper presents a new technique to help developers understand heap referencing properties of object-oriented programs and how the actions of the program affect those properties. Our thesis is that each object's referencing relationships with other objects determine important aspects of its purpose in the computation, and that we can use these referencing relationships to synthesize a set of conceptual object states (we call each state a *role*) that captures these aspects. As the program manipulates objects and changes their referencing relationships, each object transitions through a sequence of roles, with each role capturing the functionality inherent in its current referencing relationships.

We have built two tools that enable a developer to use roles to explore the behavior of object-oriented programs: 1) a dynamic role analysis tool that automatically extracts the different roles that objects play in a given computation and characterizes the effect of program actions on these roles, and 2) a graphical, interactive exploration tool that presents this information in an intuitive form to the developer. By allowing the developer to customize the presentation of this information to show the amount of detail appropriate for the task at hand, these tools support the exploration of both detailed properties within a single data structure and larger properties that span multiple data structures. Our experience using these tools indicates that they can provide substantial insight into the structure, behavior, and key properties of the program and the objects that it manipulates.

1.1 Role Separation Criteria

The foundation of our role analysis system is a set of criteria (the *role separation criteria*) that the system uses to separate instances of the same class into different roles. Conceptually, we frame the role separation criteria as a set of predicates that classify objects into roles. Each predicate captures some aspect of the object's referencing relationships. Two objects play the same role if they have the same values for these predicates. Our system supports predicates that capture the following kinds of relationships:

- **Heap Alias Relationships:** The functionality of an object often depends on the objects that refer to it. For example, instances of the `PlainSocketImpl` class acquire input and output capabilities when referred to by a `SocketInputStream` or `SocketOutputStream` object. The role separation criteria capture these distinctions by placing objects with different kinds of heap aliases in different roles. Formally, there is a role separation predicate for each field of each class. An object satisfies the predicate if one such field refers to it.
- **Reference-To Relationships:** The functionality of an object often depends on the objects to which it refers. A Java `Socket` object, for example, does not support communication until its file descriptor field

*This research was supported in part by a fellowship from the Fannie and John Hertz Foundation, DARPA Contract F33615-00-C-1692, NSF Grant CCR00-86154, and NSF Grant CCR00-63513.

ICSE '02 Orlando, Florida

refers to an actual file descriptor object. To capture these distinctions, our role separation criteria place objects in different roles if they differ in which fields contain null values. Formally, there is a predicate for each field of every class. An instance of that class satisfies the predicate if its field is not null.

- **Reachability:** The functionality of an object often depends on the specific data structures in which it participates. For example, a program may maintain two sets of objects: one set that it has completed processing, and another that it has yet to process. To capture such distinctions, our role separation criteria identify the roots of different data structures and place objects with different reachability properties from these roots in different roles. Formally, there is a predicate for each variable that may be a root of a data structure. An object satisfies the predicate if it is reachable from the variable. Additionally, we define a unique *garbage* role for unreachable objects.

- **Identity:** To facilitate navigation, data structures often contain reverse pointers. For example, the objects in a circular doubly-linked list satisfy identity predicates corresponding to the paths `next.prev` and `prev.next`. Formally, there is a role separation predicate for each pair of fields. The predicate is true if the path specified by the two fields exists and leads back to the original object.

- **History:** In some cases, objects may change their conceptual state when a method is invoked on them, but the state change may not be visible in the referencing relationships. For example, the native method `bind` assigns a name to instances of the Java `PlainSocketImpl` class, enabling them to accept connections. But the data structure changes associated with this change are hidden behind the operating system abstraction. To support this kind of conceptual state change, the role separation criteria include part of the method invocation history of each object. Formally, there is a predicate for each parameter of each method. An object satisfies one of these predicates if it was passed as that parameter in some invocation of that method.

1.2 Role Subspaces

To allow the developer to customize the role separation criteria, our system supports *role subspaces*. Each role subspace contains a subset of the possible role separation criteria. When operating within a given subspace, the tools coarsen the separation of objects into roles by eliminating any distinctions made only by criteria not in that subspace. Developers may use subspaces in a variety of ways:

- **Focused Subspaces:** As developers explore the behavior of the program, they typically focus on different and changing aspects of the object properties and referencing relationships. By choosing a subspace that excludes irrelevant criteria, the developer can explore relevant properties at an appropriate level of detail while ignoring distracting distinctions that are currently irrelevant.

- **Orthogonal Subspaces:** Developers can factor the role separation criteria into orthogonal subspaces. Each subspace identifies a current role for each object; when combined, the subspaces provide a classification structure in which each object can simultaneously play multiple roles, with each role chosen from a different subspace.

- **Hierarchical Subspaces:** Developers can construct a hierarchy of role subspaces, with child subspaces augmenting parent subspaces with additional role separation criteria. In effect, this approach allows developers to identify an increasingly precise and detailed dynamic classification hierarchy for the roles that objects play during their lifetimes in the computation.

Role subspaces give the developer great flexibility in exploring different perspectives on the behavior of the program. Developers can use subspaces to view changing object states as combinations of roles from different orthogonal role subspaces, as paths through an increasingly detailed classification hierarchy, or as individual points in a constellation of relevant states. Unlike traditional structuring mechanisms such as classes, roles and role subspaces support the evolution of multiple complementary views of the program's behavior, enabling the developer to seamlessly flow through different perspectives as he or she explores different aspects of the program at hand.

1.3 Contributions

This paper makes the following contributions:

- **Role Concept:** It introduces the concept that object referencing relationships and method invocation histories capture important aspects of an object's state, and that these relationships and histories can be used to synthesize a cognitively tractable abstraction for understanding the changing roles that objects play in the computation.

- **Role Separation Criteria:** It presents a set of criteria for classifying instances of the same class into different roles. It also presents an implemented tool that uses these criteria to automatically extract information about the roles that objects play.

- **Role Subspaces:** It shows how developers can use role subspaces to structure their understanding and presentation of the different aspects of the program state. Specifically, the developer can customize the role subspaces to focus the role separation criteria to hide (currently) irrelevant distinctions, to factor the object state into orthogonal components, and to develop object classification hierarchies.

- **Graphical Role Exploration:** It presents a tool that graphically and interactively presents role information. Specifically, this tool presents role transition diagrams, which display the trajectories that objects follow through the space of roles, and role relationship diagrams, which display referencing relationships between objects that play different roles. These diagrams are hyperlinked for easy navigation.

- **Role Exploration Strategy:** It presents a general strategy that we developed to use the tools to explore the behavior of object-oriented programs.

- **Experience:** It presents our experience using our tools on several Java programs. We found that the tools enabled us to quickly discover and understand important properties of these programs.

2. EXAMPLE

We next present a simple example that illustrates how a developer can use our tools to explore the behavior of a web server. We use a version of JhttpServer, a web server written in Java. This program accepts incoming requests for files from web browsers and serves the files back to the web browsers.

The code in the `JhttpServer` class first opens a port and waits for incoming connections. When it receives a connection, it creates a `JhttpWorker` object, passes the Socket controlling the communication to the `JhttpWorker` initializer, and turns control over to the `JhttpWorker` object.

The code in the `JhttpWorker` class first builds input and output streams corresponding to the Socket. It then parses the web browser's request to obtain the requested filename and the http version from the web browser. Next, it processes the request. Finally, it closes the streams and the socket and returns to code in the `JhttpServer` class.

2.1 Starting Out

To use our system, the developer first compiles the program using our compiler, then runs the program. The compiler inserts instrumentation code that generates an execution trace. The analysis tool then reads the trace to extract the information and convert it into a form suitable for interactive graphical display. The graphical user interface runs in a web browser with related information linked for easy navigation.

The analysis evaluates the roles of the objects at method boundaries. Our system uses four abstractions to present the observed role information to the developer: 1) role transition diagrams, which present the observed role transitions for instances of a given class, 2) role relationship diagrams, which present referencing relationships between objects from different classes, 3) role definitions, which present the referencing relationships that define each role, and 4) enhanced method interfaces, which show the object referencing properties at invocation and the effect of the method on the roles of the objects that it accesses.

2.2 Role Transition Diagrams

Developers typically start exploring the behavior of a program by examining role transition diagrams to get a feel for the different roles that instances of each class play in the computation. In this example, we assume the developer first examines the role transition diagram for the `JhttpWorker` class, which handles client requests. Figure 1 presents this diagram.[1] The ellipses represent roles and the arrows represent transitions between roles. Each arrow is labeled with the method that caused the object to take the transition. Solid edges denote the execution of methods that take the `JhttpWorker` as a parameter; dotted edges denote portions of a method or methods that change the roles of `JhttpWorker` objects, but do not take the `JhttpWorker` object as a parameter. The diagram always presents the most deeply nested (in the call graph) method responsible for the role change.

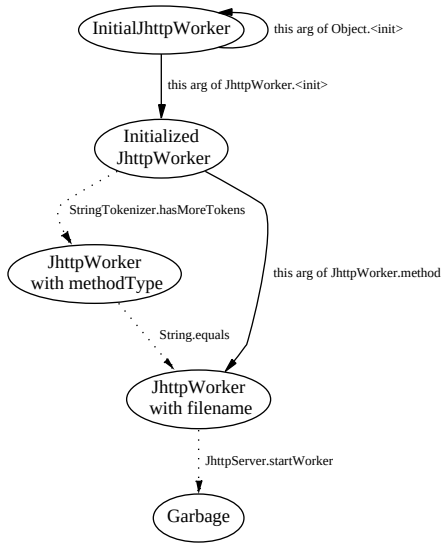

Figure 1: Role transition diagram for `JhttpWorker` class

2.3 Role Definitions

Role transition diagrams show how objects transition between roles, but provide little information about the roles themselves. Our graphical interface therefore links each role node with its *role definition*, which specifies the properties that all objects playing that role must have. Figure 2 presents the role definition for the JhttpWorker with filename role, which is easily accessible by using the mouse to select the role's node in the role transition diagram. This definition specifies that instances of the JhttpWorker with filename role have the class `JhttpWorker`, no heap aliases, no identity relations, and references to heap objects in the fields `httpVersion`, `fileName`, `methodType`, and `client`.

```
Role: JhttpWorker with filename
  Class: JhttpWorker
  Heap aliases: none
  non-null fields: httpVersion, fileName,
                   methodType, client
  identity relations: none
```

Figure 2: Sample role definition for `JhttpWorker` class

2.4 Role Relationship Diagrams

After obtaining an understanding of the roles of important classes, the developer typically moves on to consider relationships between objects of different classes. These relationships are often crucial for understanding the larger data

[1] In addition to graphically presenting these diagrams in a web browser, our tool is capable of generating PostScript images of each diagram using the dot tool [1]. Our tool automatically generates initial names for roles and allows the developer to rename the roles. All of the diagrams presented in this paper were generated automatically from our tool with renaming in some cases for clarification.

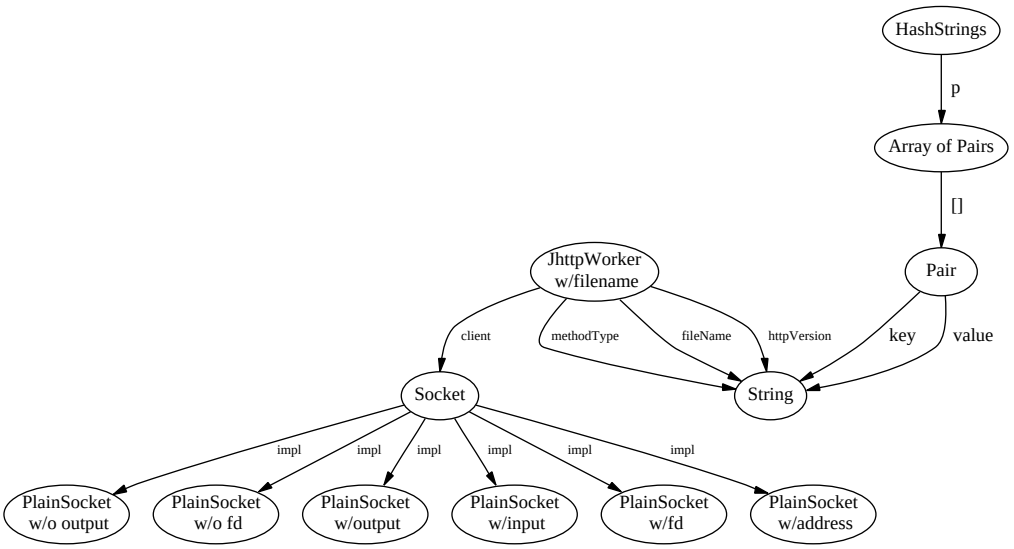

Figure 3: Portion of role relationship diagram for JhttpServer

structures that the program manipulates. Role relationship diagrams are the primary tool that developers use to help them understand these relationships. Figure 3 presents a portion of the role relationship diagram surrounding one of the roles of the `JhttpWorker` class. The ellipses in this diagram represent roles, and the arrows represent referencing relationships between objects playing those roles.

Note that some of the groups of roles presented in Figure 3 correspond to combinations of objects that conceptually act as a single entity. For example, the `HashStrings` object and the underlying array of `Pairs` that it points to implement a map from `String` to `String`. Developers often wish to view a less detailed role relationship diagram that merges the roles for these kinds of combinations.

In many cases, the analysis can automatically recognize these combinations and represent them with a single role node. Figure 4 presents the role relationship diagram that the tool produces when the developer turns this option on. Notice that the analysis recognizes the `Socket` object and the `httpVersion` string as being part of the `JhttpWorker` object. Also notice that it recognizes the `Pair` arrays, `Pair` objects, and key strings as being part of the corresponding `HashStrings` object, with the key strings disappearing in the abstracted diagram because they are encapsulated within the `HashStrings` data structure. The analysis allows the developer to choose, for each class, a policy that determines how (and if) the analysis merges roles of that class into larger data structures.

An examination of Figures 3 and 4 shows that instances of the `PlainSocketImpl` class play many different roles. To explore these roles, the developer examines the role transition diagram for the `PlainSocketImpl` class. Figure 5 presents this diagram. The diagram contains two disjoint sets of roles, each branching off of the Initial PlainSocket role. This structure indicates that instances of the class have two distinct purposes in the computation. Some instances manage communication over a TCP/IP connection, while others accept incoming connections.

```
Method: SocketInputStream.<init>(this,plainsocket)
 Call Context: {
  this: Initial InputStream -> InputStream w/impl,
  plainsocket: PlainSocket w/fd ->
    PlainSocket w/input }
 Write Effects:
  this.impl=plainsocket
  this.temp=NEW
  this.fd=plainsocket.fd
 Read Effects:
  plainsocket
  NEW
  plainsocket.fd
 Role Transition Effects:
  plainsocket: PlainSocket w/fd -> PlainSocket
                                   w/input
  this: Initial InputStream -> InputStream w/fd
  this: InputStream w/fd -> InputStream w/impl
```

Figure 6: Enhanced Method Interface for `SocketInputStream` initializer

2.5 Enhanced Method Interfaces

Finally, our tool can present information about the roles of parameters and the effect of each method on the roles that different objects play. Given a method, our tool presents this information in the form of an enhanced method interface. This interface provides the roles of the parameters at method entry and exit and any read, write, or role transition effects the method may have. Figure 6 presents an enhanced method interface for the `SocketInputStream` initializer. This interface indicates that the `SocketInputStream` initializer operates on objects that play the roles Initial InputStream and PlainSocket w/fd. When it executes, it changes the roles of these objects to InputStream w/impl and PlainSocket w/input, respectively.

Enhanced method interfaces provide the developer with additional information about the (otherwise implicit) as-

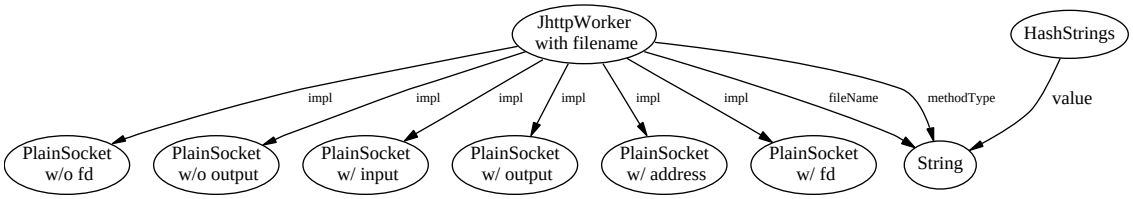

Figure 4: Portion of role relationship diagram for JhttpServer after part object abstraction

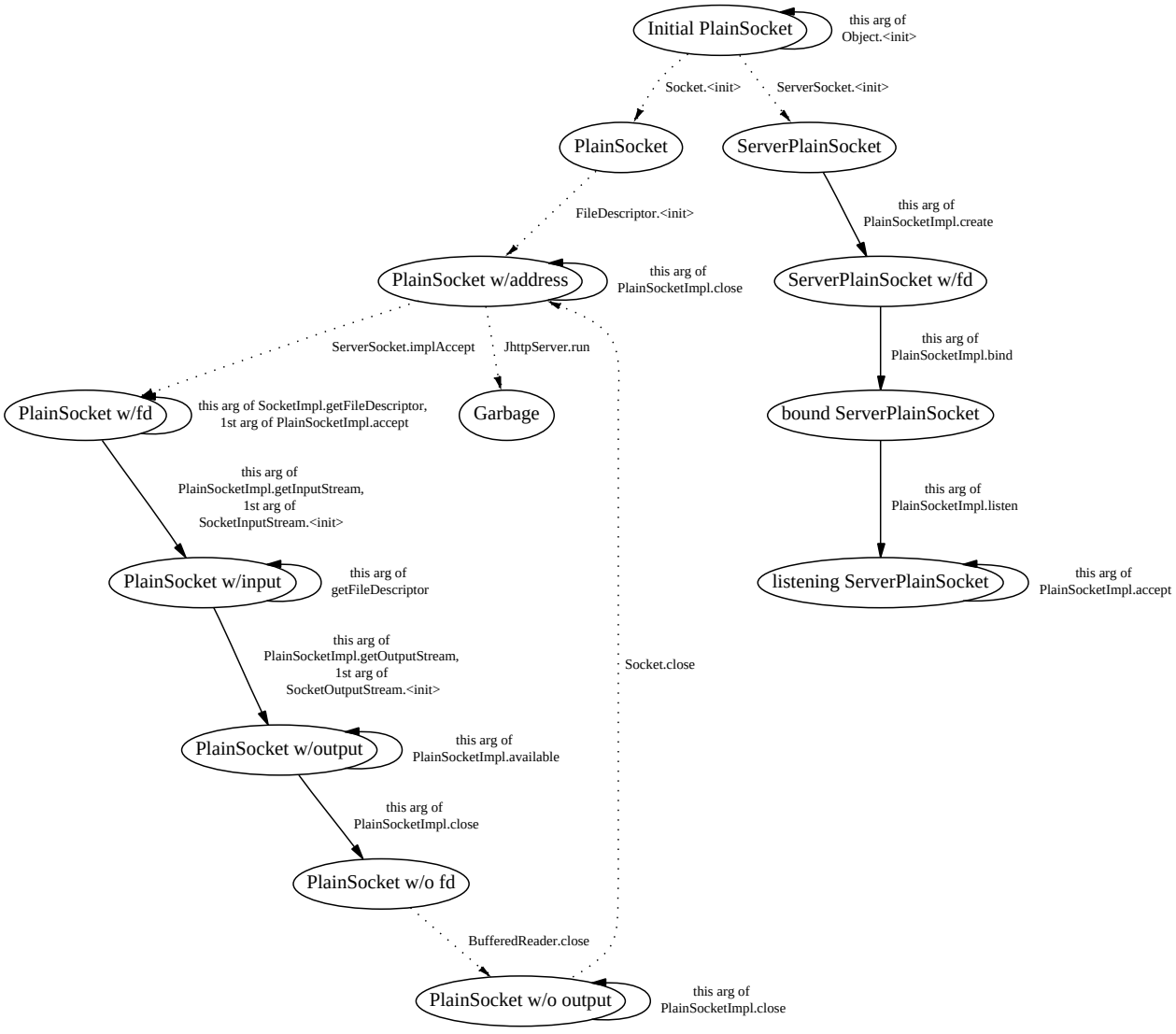

Figure 5: Role transition diagram for the `PlainSocketImpl` class

sumptions that the method may make about its parameters and the roles of the objects that it manipulates. This information may help the developer better understand the purpose of the method in the computation and provide guidelines for its successful use in other contexts.

2.6 Role Information

In general, roles capture important properties of the objects and provide useful information about how the actions of the program affect those properties.

- **Consistency Properties:** Our analysis can discover program-level data structure consistency properties.
- **Enhanced Method Interfaces:** In many cases, the interface of a method makes assumptions about the referencing relations of its parameters. Our analysis can discover constraints on the roles of parameters of a method and determine the effect of the method on the heap.
- **Multiple Uses:** Code factoring minimizes code duplication by producing general-purpose classes (such as the Java `Vector` and `Hashtable` classes) that can be used in a variety of contexts. But this practice obscures the different purposes that different instances of these classes serve in the computation. Our analysis can rediscover these distinctions.
- **Correlated Relationships:** In many cases, groups of objects cooperate to implement a piece of functionality, with the roles of the objects in the group changing together over the course of the computation. Our analysis can discover these correlated state changes.

3. DYNAMIC ANALYSIS

We implemented the dynamic analysis as several components. The first component uses the MIT FLEX compiler [2] to instrument Java programs to generate execution traces. Because this component operates on Java bytecodes, our system does not require source code. The instrumented program assigns unique identifiers to every object and reports relevant heap and pointer operations in the execution trace. The second component uses the trace to reconstruct the heap. As part of this computation, it also calculates reachability information and records the effect of each method's execution on the roles of the objects that it manipulates.

3.1 Predicate Evaluation

The dynamic analysis uses the information it extracts from the trace to apply the role separation criteria as follows:

- **Heap Aliases:** In addition to reconstructing the heap, the analysis also maintains a set of *inverse references*. There is one inverse reference for each reference in the original heap. For each reference to a target object, the inverse reference enables the dynamic analysis to quickly find the source of the reference and the field containing the reference. To compute the heap alias predicates for a given object, the analysis examines the inverse references for that object.
- **Reference-To:** The reconstructed heap contains all of the references from the original program, enabling the analysis to quickly compute all of the reference-to predicates for a given object by examining its list of references.
- **Identity:** To compute the identity predicates for a given object, the analysis traces all paths of length two from the object to find paths that lead back to the object.
- **Reachability:** There are two key issues in computing the reachability information: using an efficient incremental reachability algorithm and choosing the correct set of variables to include in the role separation criteria. Whenever the program changes a reference, the incremental reachability algorithm finds the object whose reachability properties may have changed, and then incrementally propagates the reachability changes through the reconstructed heap.

  To avoid undesirable separation caused by an inappropriate inclusion of temporary variables into the role separation criteria, our implemented system uses two rules to identify variables that are the roots of data structures. If an object o is reachable from variables x and y that point to objects o_x and o_y respectively, and o_x is reachable from y but o_y is not reachable from x, then we exclude x from the role separation criteria. Alternatively, if o_x is reachable from y, o_y is reachable from x, and the reference y was created before the reference x, we exclude x from the criteria.

  These rules keep temporary references used for traversing heap structures from becoming part of the role definitions, but allow long term references to the roots of data structures to be incorporated into role definitions. These rules also have the property that if an object is included in two disjoint data structures with different roots, then the object's role will reflect this double inclusion.
- **Method Invocation History:** Whenever an object is passed as a parameter to a method, the analysis records the invocation as part of the object's method invocation history. This record is then used to evaluate method invocation history predicates when assigning future roles to the object.
- **Array Roles:** We treat arrays as objects with a special `[]` field, which points to the elements of the array. Additionally, we generalize the treatment of reference-to relations to allow roles to specify the classes and the corresponding number (up to some bound) of the array's elements.

By default, the analyzer evaluates these predicates at every method entry and exit point. We allow the developer to coarsen this granularity by declaring methods *atomic*, in which case the analysis attributes all role transitions that occur inside the method to the method itself. This is implemented by not checking for role transitions until the atomic method returns. This mechanism hides temporary or irrelevant role transitions that occur inside the method. This feature is most useful for simplifying role transition diagrams. In particular, many programs have a complicated process

[2]Available at www.flexc.lcs.mit.edu.

for initializing objects. Once we use the role transition diagram to understand this process, we often find it useful to abstract the entire initialization process as atomically generating a fully initialized object.

3.2 Multiple Object Data Structures

A single data structure often contains many component objects. Java `HashMap` objects, for example, use an array of linked lists to implement a single map. To enable the developer to view such composite data structures as a single entity, our dynamic analysis supports operations that merge multiple objects into a single entity. Specifically, the dynamic analysis can optionally recognize any object playing a given role (such roles are called *part roles*) as conceptually part of the object that refers to it. The user interface will then merge all of the role information from the part role into the role of the object that refers to it.

Depending on the task at hand, different levels of abstraction may be useful to the developer. On a per class basis, the developer can specify whether to merge one object's role into another object's role. The analysis provides four different policies: never merge, always merge, merge only if one heap reference to the object ever exists, and merge only if one heap reference at a time exists to the object. The analysis implements these policies using a two pass strategy: one pass identifies concrete objects that meet the merging criterion, and another assigns the selected objects part roles. The analysis requires that any cycles in the heap include at least one object that does not have a part role.

3.3 Method Effect Inference

For each method execution, the dynamic analysis records the reads, writes, and role transitions that the execution performs. Each method effect summary uses regular expressions to identify paths to the accessed or affected objects. These paths are identified relative to the method parameters or global variables and specify edges in the heap that existed when the method was invoked. Method effect inference therefore has two steps: detecting concrete paths with respect to the heap at procedure invocation and summarizing these paths into regular expressions.

To detect concrete paths, we keep a path table for each method invocation. This table contains the concrete path, in terms of the heap that existed when the method was invoked, to all objects that the execution of the method may affect. At method invocation, our analysis records the objects to which the parameters and the global variables point. Whenever the execution retrieves a reference to an object or changes an object's reachability information, the analysis records a path to that object in the path table. If the execution creates a new object, we add a special NEW token to the path table; this token represents the path to that object.

We obtain the regular expressions in the method effect summary by applying a set of rewrite rules to the extracted concrete paths. Figure 7 presents the current set of rewrite rules. Given a concrete path $f_1.f_2...f_n$, we apply the rewrite rules to the tuple $\langle \epsilon, f_1.f_2...f_n \rangle$ to obtain a final tuple $\langle Q, \epsilon \rangle$, where Q is the regular expression that represents the path. We present the rewrite rules in the order in which they are applied. We use the notation that $\kappa(f)$ denotes the class in which the field f is declared as an instance variable, and $\tau(f)$ is the declared type of the field f.

Rules 1 and 2 simplify intermediate expressions generated during the rewrite process. Rules 3 and 4 generalize concrete paths involving similar fields such as paths through a binary tree. Rules 5 and 6 generalize repeated sequences in concrete paths. The goal is to capture paths generated in loops or recursive methods and ensure that path expressions are not overly specialized to any particular execution.

1. $\langle Q.(q_1...(e_1 \mid f \mid e_2 \mid f \mid e_3)...q_n)^*, Q' \rangle \Rightarrow$
   $\langle Q.(q_1...(e_1 \mid f \mid e_2 \mid e_3)...q_n)^*, Q' \rangle$

2. $\langle Q.(q_1...(e_1 \mid f \mid e_2 \mid f \mid e_3)^*...q_n)^*, Q' \rangle \Rightarrow$
   $\langle Q.(q_1...(e_1 \mid f \mid e_2 \mid e_3)^*...q_n)^*, Q' \rangle$

3. $\langle Q.(f_1), f_2.Q' \rangle \Rightarrow \langle Q.(f_1 \mid f_2)^*, Q' \rangle$
   if $\kappa(f_1) = \kappa(f_2)$ and $\tau(f_1) = \tau(f_2)$

4. $\langle Q.(f_0 \mid ... \mid f_n)^*, f'.Q' \rangle \Rightarrow \langle Q.(f_0 \mid ... \mid f_n \mid f')^*, Q' \rangle$
   if $\kappa(f_n) = \kappa(f')$ and $\tau(f_n) = \tau(f')$

5. $\langle Q.q_1...q_n.q'_1...q'_n, Q' \rangle \Rightarrow \langle Q.(q_1 \oplus q'_1...q_n \oplus q'_n)^*, Q' \rangle$
   if $\forall i, 1 \leq i \leq n, q_i \equiv q'_i$, where $q \equiv q'$ if
   (a) $q = (f_1 \mid ... \mid f_j), q' = (f'_1 \mid ... \mid f'_k)$,
   $\kappa(f_1) = \kappa(f'_1)$ and $\tau(f_1) = \tau(f'_1)$, or
   (b) $q = (f_1 \mid ... \mid f_j)^*, q' = (f'_1 \mid ... \mid f'_k)^*$,
   $\kappa(f_1) = \kappa(f'_1)$ and $\tau(f_1) = \tau(f'_1)$.
   $(f_1 \mid ... \mid f_j) \oplus (f'_1 \mid ... \mid f'_k) = (f_1 \mid ... \mid f_j \mid f'_1 \mid ... \mid f'_k)$
   $(f_1 \mid ... \mid f_j)^* \oplus (f'_1 \mid ... \mid f'_k)^* =$
   $(f_1 \mid ... \mid f_j \mid f'_1 \mid ... \mid f'_k)^*$

6. $\langle Q.(q_1...q_n)^*.q'_1...q'_n, Q' \rangle \Rightarrow \langle Q.(q_1 \oplus q'_1...q_n \oplus q'_n)^*, Q' \rangle$
   if $\forall i, 1 \leq i \leq n, (q_i \equiv q'_i)$.

7. $\langle Q, f.Q' \rangle \Rightarrow \langle Q.(f), Q' \rangle$

Figure 7: Rewrite rules for paths

For read or role transition effects, we record the starting point and regular expression for the path to the object. For write effects, we give the starting points for both objects and the regular expressions for the paths. Valid starting points are method parameters and global variables. We denote effects for objects created in a procedure using the NEW token. We denote writing a null pointer to an object's field using the NULL token.

3.4 Role Subspaces

Our tool allows the developer to define multiple role subspaces and modify the role separation criteria for each subspace as follows:

- **Fields:** The developer can specify fields to ignore for the purpose of assigning roles. The analysis will show these fields in the role relationship diagram, but the references in these fields will not affect the roles assigned to the objects.
- **Methods:** The developer can specify which methods and which parameters to include in the role separation criteria.
- **Reachability:** The developer can specify variables to include or to exclude from the reachability-based role separation criteria.

- **Classes:** The developer can collapse all objects of a given class into a single role.

In practice, we have found role subspaces both useful and usable — useful because they enabled us to isolate the important aspects of relevant parts of the system while eliminating irrelevant and distracting detail in other parts, and usable because we were usually able to obtain a satisfactory role subspace with just a small number of changes to the default criteria.

4. USER INTERFACE

The user interface presents four kinds of web pages: class pages, role pages, method pages, and the role relationship page. Each class page presents the role transition diagram for the class. From the class page, the developer can click on the nodes and edges in the role transition diagram to see the corresponding role and method pages for the selected node or edge. Each role page presents a role definition, displaying related roles and classes and enabling the developer to select these related roles and classes to bring up the appropriate role or class page. Each method page shows the developer which methods called the given method and allows the developer to configure method-specific abstraction policies. The role relationship page presents the role relationship diagram. From this diagram, the developer can select a role node to see the appropriate role definition page.

The user interface allows the developer to create and manipulate multiple role subspaces. The developer can create a new role subspace by selecting a set of predicates to determine the role separation criteria, then combine subspaces to define views. Views with a single subspace use the role separation criteria from that subspace. Views with multiple subspaces use a cross product operator to combine the roles from the different subspaces, with the final set of roles isomorphic to those obtained by taking the union of the role separation criteria from all of the subspaces. Within a view, the developer can identify additional role subspaces to be used for labeling purposes. These role subspaces do not affect the separation of objects into roles, but rather label each role in the view with the roles that objects playing those roles have in these additional labeling subspaces.

5. EXPLORATION STRATEGY

As we used the tool, we developed the following strategy for exploring the behavior of a new program. We believe this strategy is useful for structuring the process of using the tool, and that most developers will use some variant of this strategy.

When we started using the tool on a new program, we first recompiled the program with our instrumentation package, and then ran the program to obtain an execution trace. We then used our graphical tool to browse the role transition diagrams for each of the classes, looking for interesting initialization sequences, splits in the role transition diagram indicating different uses for objects of the class, and transition sequences indicating potential changes in the purpose of instances of the class in the computation.

During this activity, we were interested in obtaining a broad overview of the actions of the program. We therefore often found opportunities to appropriately simplify the role transition diagrams, typically by creating a role subspace to hide irrelevant detail, by declaring initializing methods atomic, or by utilizing the multiple object abstraction feature. Occasionally, we found opportunities to include aspects of the method invocation history into the role separation criteria. We found that our default policy for merging multiple object data structures into a single data structure for role presentation purposes worked well during this phase of the exploration process.

Once we had created role subspaces revealing roles at an appropriate granularity, we then browsed the enhanced method interfaces to discover important constraints on the roles of the objects passed as parameters to the method. This information enabled us to better understand the correlation between the actions of the method and the role transitions, helping us to isolate the regions of the program that performed important modifications, such as insertions or removals from collections. It also helped us understand the (otherwise implicit) assumptions that each method made about the states of its parameters. We found this information useful in understanding the program; we expect maintainers to find it invaluable.

We next observed the role relationship diagram. This diagram helped us to better understand the relationships between classes that work together to implement a given piece of functionality. In general, we found that the complete role relationship diagram presented too much information for us to use it effectively. We therefore adopted a strategy in which we identified a starting class of interest, then viewed the region surrounding the roles of that class. We found that this strategy enabled us to quickly and effectively find the information we needed in the role relationship diagram.

Finally, we sometimes decided to explore several roles in more detail. We often returned to the role transition diagram and created a customized role subspace to expose more detail for the current class but less detail for less relevant classes. In effect, this activity enabled us to easily adapt the system to view the program from a more specialized perspective. Given our experience using this feature of our role analysis tool, we believe that this ability will prove valuable for any program understanding tool.

6. EXPERIENCE

We next discuss our experience using our role analysis tool to explore the behavior of several Java programs. We report our experience for several programs: Jess, an expert system shell in the SpecJVM benchmark suite; Direct-To, a Java version of an air-traffic control tool; Tagger, a text formatting program; Treeadd, a tree manipulation benchmark in the J. Olden benchmark suite [3]; and Em3d, a scientific computation in the J. Olden benchmark suite.

6.1 Jess

Jess first builds a network of nodes, then performs a computation over this network. While the network contains many different kinds of nodes, all of the nodes exhibit a similar construction and use pattern. Consider, for example, instances of the `Node1TELN` class. Figure 8 presents the role transition diagram for objects of this class. An examination of this diagram and the linked role definitions shows that during the construction of the network, the program represents the edges between nodes using a resizable vector of references to `Successor` objects, each of which is a wrapper

[3]Available at www-ali.cs.umass.edu/~cahoon.

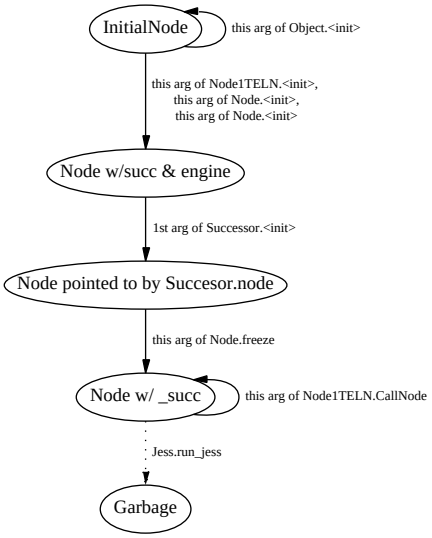

Figure 8: Role transition diagram for the `Node1TELN` class

around a node object. The `succ` field refers to this vector. When the network is complete, the program constructs a less flexible but more efficient representation in which each node contains a fixed-size array of references to other nodes; the `_succ` field refers to this array. This change occurs when the program invokes the `freeze` method on the node. All of the nodes in the program exhibit this construction pattern.

The generated method annotations provide information about the assumptions that several key methods make about the roles of their parameters. Specifically, these annotations show that the program invokes the `CallNode` method (this method implements the primary computation on the network) on a node only after the `freeze` method has converted the representation of the edges associated with the node to the more efficient form.

The role definitions also provide information about network's structure, specifically that all of the nodes in the network have either one or two incoming edges. Each fully constructed instance of the `Node1TELN`, `Node1TECT`, `Node1TEQ`, `NodeTerm`, or `Node1TMF` class has exactly one `Successor` object that refers to it, indicating that these kinds of nodes all have exactly one incoming edge. Each fully constructed instance of the `Node2` class, on the other hand, has exactly two references from `Successor` objects, indicating that `Node2` nodes have exactly two incoming edges.

6.2 Direct-To

Direct-To is a prototype Java implementation of a component of the Center-Tracon Automation System (CTAS) [7]. The tool helps air-traffic controllers streamline flight paths by eliminating intermediate points; the key constraint is that these changes should not cause new conflicts, which occur when aircraft pass too close to each other.

We first discuss our experience with the `Flight` class, which represents flights in progress. Each `Flight` object contains references to other objects, such as `FlightPlan` objects and `Route` objects, that are part of its state. Our analysis recognized these other objects as part of the corresponding `Flight` object's state, and merged all of these objects into a single multiple object data structure.

Roles helped us understand the initialization sequence and subsequent usage pattern of Flight objects. An initialized `Flight` object has been inserted into the flight list; various fields of the object refer to the objects that implement the flight's identifier, type, aircraft type, and flight plan. Once initialized, the flight is ready to participate in the main computation of the program, which repeatedly acquires a radar track for the flight and uses the track and the flight plan to compute a projected trajectory. The initialization sequence is clearly visible in the role transition diagram, which shows a linear sequence of role transitions as the flight object acquires references to its part objects and is inserted into the list of flights. The acquisition and computation of the tracks and trajectories also show up as transitions in this diagram.

Roles also enabled us to untangle the different ways in which the program uses instances of the `Point4d` class. Specifically, the program uses instances of this class to represent aircraft tracks, trajectories, and velocities. The role transition diagram makes these different uses obvious: each use corresponds to a different region of roles in the diagram. No transitions exist between these different regions, indicating that the program uses the corresponding objects for disjoint purposes.

6.3 Tagger

Tagger is a document layout tool written by Daniel Jackson. It processes a stream of text interspersed with tokens that identify when conceptual components such as paragraphs begin and end. Tagger works by first attaching action objects to each token, and then processing the text and tokens in order. Whenever it encounters a token, it executes the attached action.

It turns out that there are dependences between the operations of the program and the roles of the actions and tokens. For example, one of the tokens causes the output of the following paragraph to be suppressed. Tagger implements this suppression action with pairs of matched suppress/unsuppress actions. When the suppress action executes, it places an unsuppress action at the end of the paragraph, ensuring that only one paragraph will be suppressed. These actions are reflected in role transitions as follows. When the program binds the suppress action to a token, the action takes a transition because of the reference from the token. When the suppress action executes, it binds the corresponding unsuppress action to the token at the end of the paragraph, causing the unsuppress action to take a transition to a new state. Roles therefore enabled us to discover an interesting correlation between the execution of the suppress action and data structure modifications required to undo the action later. We were also able to observe a role-dependent interface — the method that executes actions always executes actions that are bound to tokens.

6.4 Treeadd

Treeadd builds a tree of `TreeNode` objects; each such object has an integer value field. It then calculates the sum of the values of the nodes. The role analysis tool extracted some interesting properties of the data structure and gave us insight into the behavior of the parts of the program that construct and use the tree.

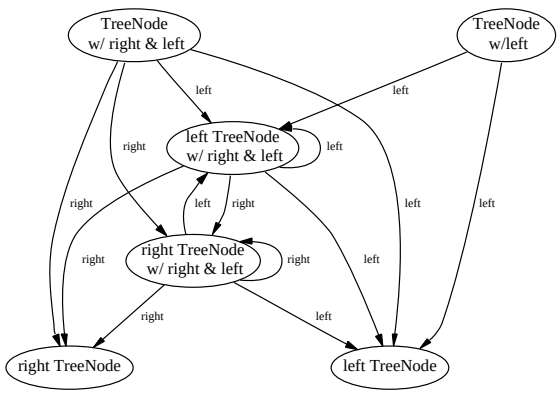

Figure 9: Role relationship diagram for the `TreeNode` class

Figure 9 presents the region of the role relationship diagram that contains the roles of `TreeNode` objects. By examining this diagram and the linked role definitions, we were able to determine that the `TreeNode` objects did in fact comprise a tree — the roles corresponding to the root of the tree have no references from `left` or `right` fields of other `TreeNode` objects, and all other `TreeNode` roles have exactly one reference from the `left` or `right` field of another `TreeNode`.

Figure 10 presents the role transition diagram for `TreeNode` objects. This diagram, in combination with the linked role definitions, clearly shows a bottom-up initialization sequence in which each `TreeNode` acquires a left child and a right child, then a reference from the `right` or `left` field of its parent. Alternative initialization sequences produce `TreeNode` objects with no children. Note that the automatically generated role names in this figure are intended to help the developer understand the referencing relationships that define each role. The role name Right TreeNode w/right & left, for example, indicates that objects playing the role have 1) a reference from the `right` field of an object, and 2) non-null `right` and `left` fields. The role name TreeNode w/left indicates that an object playing this role has a non-null `left` field.

6.5 Em3d

Em3d simulates the propagation of electromagnetic waves through objects in three dimensions. It uses enumerators extensively in two phases of the computation. The first phase builds a graph that models the electric and magnetic fields; the second phase traverses the graph to simulate the propagation of these fields. The role transition diagram for the enumerator objects contains roles corresponding to an initialized enumerator, an enumerator with remaining elements, and an enumerator with no remaining elements. As expected, the program never invokes the `next` method on an enumerator object that has no remaining elements, enabling the developer to verify that the program uses enumerator objects in a standard way.

6.6 Utility of Roles

In general, roles helped us to discover key data structure properties and understand how the program initialized and manipulated objects and data structures. The combination of the role relationship diagram and linked role definitions typically provided the most useful information about data structure properties. Examples of these properties include the referencing properties of `TreeNode` objects in the Treeadd benchmark and the correspondence between `Successor` nodes and network nodes in Jess.

The role transition diagram typically provided the most useful information about object initialization sequences and usage patterns. Examples of object initialization sequences include the initialization of `Flight` objects in the Direct-to benchmark and of `TreeNode` objects in the Treeadd benchmark. Jess provides an interesting example of a conceptual phase transition in a data structure — the program uses a more flexible but less efficient data structure during a construction phase, then replaces this data structure with a more efficient frozen version for a subsequent computation phase. The `Point4d` class in Direct-to provides a good example of how a program can use instances of a single class for several different purposes in the computation. In all of these cases, the role analysis enabled us to quickly understand the underlying initialization sequences or usage patterns.

Finally, we found that the information about the roles of method parameters helped us to understand the otherwise implicit expectations that methods have about the states of their parameters and the effects of methods on these states. Examples of methods with important expectations or effects include the `freeze` and `CallNode` methods in Jess and the `next` method in Em3d. In general, we expect the role analysis tool to be useful in the software development process in the following ways:

- **Program Understanding:** Developers have to understand programs to modify or reuse them. In object-oriented languages, understanding heap allocated data structures is key to understanding the program. Roles help developers discover key data structure invariants and understand how programs initialize and manipulate these data structures, thus aiding program comprehension.

- **Maintenance:** To safely modify programs, developers need to understand the data structures these programs build, the referencing relations methods assume, and the effects of methods on these data structures. We expect that the diagrams and enhanced method interfaces that our tool generates will prove useful for this purpose.

- **Verifying Expected Behavior:** Developers can use our tool as a debugging aid. Developers write programs with certain invariants about heap structures in mind. If the role relationships our tool discovers are inconsistent with these invariants, the developer knows that a bug exists. Finally, the enhanced method interfaces and role transition diagrams can help the developer quickly isolate the bug.

- **Documentation:** Developers often need to document high-level properties of the program. Roles may provide an effective documentation mechanism, because

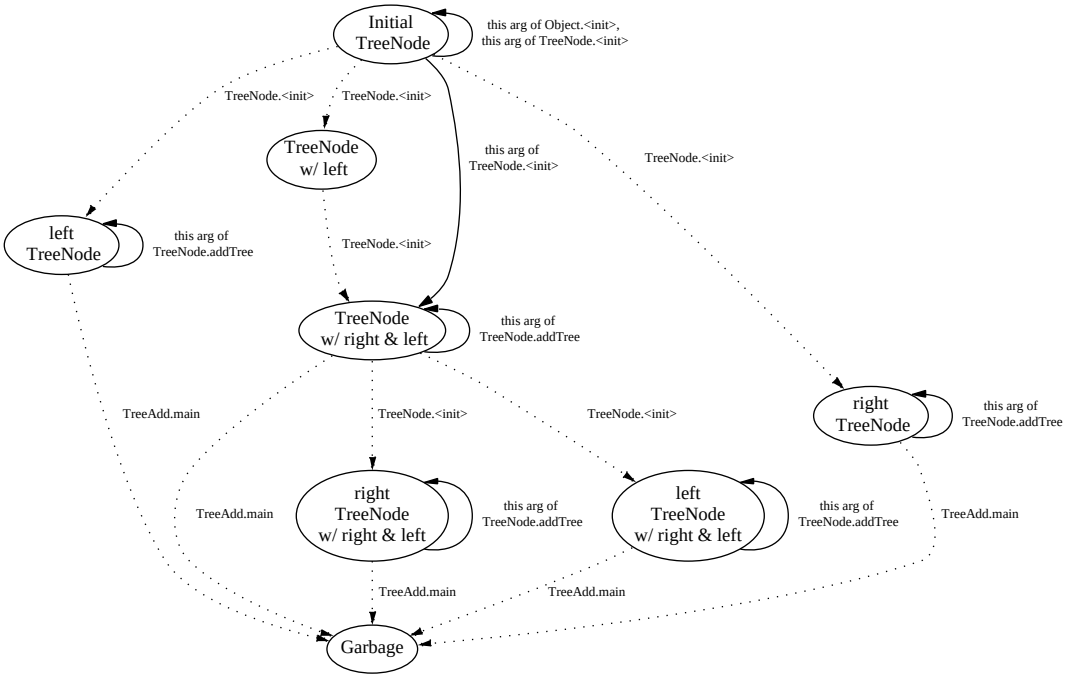

Figure 10: Role transition diagram for the `TreeNode` class

they come with a set of appealing interactive graphical representations, because they can often capture key properties of the program in a concise, cognitively tractable representation, and because (at least for the roles that our analysis tool discovers) they are guaranteed to faithfully reflect some of the behaviors of the program. Role subspaces may prove to be especially useful in presenting focused, orthogonal, or hierarchical perspectives on the purposes of the objects in the program.

- **Design:** High-level design formalisms often focus on the conceptual states of objects and the relationships between objects in these states. Our role analysis can extract information that is often similar to this design information, helping the developer to establish the connection between the design and the behavior of the program. Furthermore, the role abstraction suggests several concrete ways of realizing high-level design patterns in the code. As developers become used to working with roles, they may very well adopt role-inspired coding styles that facilitate the verification of a guaranteed connection between the high-level design and its realization in the program.

7. RELATED WORK

We survey related work in three fields: design formalisms that involve the concept of abstract object states, program understanding tools that focus on properties of the objects that programs manipulate, and static analyses for automatically discovering or verifying properties of linked data structures.

7.1 Design Formalisms

Early design formalisms identified changes in abstract object or component states as an important aspect of the design of the program [13]. Our tool also focuses on abstract state changes as a key aspect, but uses the role separation criteria to automatically synthesize a set of abstract object states rather than relying on the developer to specify the abstract state space explicitly.

Object models enable a developer to describe relationships between objects, both at a conceptual level and as realized in programs. Object modeling languages such as UML [12] and Alloy [6] can describe the different states that objects can be in, the constraints that these states satisfy, and the transitions between these states. One can view our role analysis tool as a way of automatically extracting an object model that captures the important aspects of the objects that the program manipulates. In this sense our tool establishes a connection between the abstract concepts in the object model and the concrete realization of those concepts in the objects that the program manipulates.

The concept of objects playing different roles in the computation while maintaining their identity often arises in the conceptual design of systems [5], and researchers have proposed several methodologies for realizing these roles in the program [5, 4, 9]. Our role analysis tool can recognize many of the design patterns used to implement these roles, and may therefore help developers establish a connection between an existing conceptual system design and its realization in the program. Conversely, our role separation criteria may also suggest alternate ways to implement conceptual roles. In particular, previously proposed methodologies tend to focus on ways to tag objects with (potentially redundant)

information indicating their roles, while the role separation criteria identify data structure membership (which may not be directly observable in the state of the object itself) as an important property that helps to determine the roles that the object plays.

7.2 Program Understanding Tools

Daikon [2] extracts likely algebraic invariants from information gathered during the program's execution. For example, Daikon can infer invariants such as "$y = 2x$". Daikon handles heap structures in a limited fashion by linearizing them into arrays under some specific conditions [3]. Our work differs in that we handle heap structures in a much more general fashion and focus on referencing relationships as opposed to algebraic invariants.

Womble [8] and Chava [10] both use a static analysis to automatically extract object models for Java programs. Both tools use information from the class and field declarations; Womble also uses a set of heuristics to generate conjectures regarding associations between classes, field multiplicities, and mutability.

Unlike our role analysis tool, Womble and Chava do not support the concept of an object that changes state during the execution of the program. They instead statically group all instances of the same class into the same category of objects in the object model, ignoring any conceptual state changes that may occur because of method invocations, changes to the object referencing relationships, or reachability changes.

7.3 Verifying Data Structure Properties

The analysis presented in this paper extracts role information for a single execution of the program. While it would be straightforward to combine information from multiple executions, the tool is not designed to extract or verify role information that is guaranteed to fully characterize all executions.

Statically extracting or verifying the detailed object referencing properties that roles characterize is clearly beyond the capabilities of standard pointer analysis algorithms. Researchers in our group have, however, been able to leverage techniques from precise shape analysis algorithms to develop an augmented type system and analysis algorithm that is capable of verifying that all executions of a program respect a given set of role declarations [11]. In this context, our dynamic tool could generate candidate role declarations for existing programs. Such a candidate generation system would have to be designed carefully — we expect the dynamic role analysis to be capable of extracting properties that are beyond the verification capabilities of the static role analysis.

8. CONCLUSION

We believe that roles are a valuable abstraction for helping developers to understand the objects and data structures that programs manipulate. We have implemented a dynamic role analysis tool and a flexible interactive graphical user interface that helps developers navigate the information that the analysis produces. Our experience with several Java applications indicates that our tools can help developers discover important object initialization sequences, object usage patterns, data structure invariants, and constraints on the states and referencing relationships of method parameters. Other potential applications include documenting high-level properties of the program (and especially properties that involve orthogonal or hierarchical object and data structure classification structures), discovering correlated state changes between objects that participate in the same data structure, providing specifications for a static role analysis algorithm, verifying or refuting a debugger's hypotheses about important data structure invariants, and providing a foundation for establishing a guaranteed connection between the high-level design and its realization in the program.

Acknowledgments

We would like to thank Michael Ernst, Daniel Jackson, and Viktor Kuncak for useful feedback and discussions concerning this paper and dynamic role analysis in general. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

9. REFERENCES

[1] J. Ellson, E. Gansner, E. Koutsofios, and S. North. Graphviz. `http://www.research.att.com/sw/tools/graphviz`.

[2] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, pages 449–458, 2000.

[3] M. D. Ernst, Y. Kataoka, W. G. Griswold, and D. Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, November 1999.

[4] R. Familiar. Adaptive role playing. `http://www.ccs.neu.edu/research/demeter/adaptive-patterns/arp-bofam-checked.html`.

[5] M. Fowler. Dealing with roles. `http://www.martinfowler.com/apsupp/roles.pdf`, July 1997.

[6] D. Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, Laboratory for Computer Science, Massachusetts Institute of Technology, 2000.

[7] D. Jackson and J. Chapin. Redesigning air-traffic control: A case study in software design, 2000.

[8] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *International Conference on Software Engineering*, pages 194–202, 1999.

[9] B. Jacobs. Patterns using procedural/relational paradigm. `http://www.geocities.com/tablizer/prpats.htm`.

[10] J. Korn, Y.-F. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of Java applets. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 314–325, October 1999.

[11] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, 2002.

[12] Rational Inc. The unified modeling language. http://www.rational.com/uml.

[13] W. E. Riddle, J. Sayler, A. Segal, and J. Wileden. An introduction to the dream software design system. volume 2, pages 11–23, July 1977.