Automatic Extraction of Heap Reference Properties in Object-Oriented Programs

Brian Demsky, Martin Rinard

***Abstract*— We present a new technique for helping developers understand heap referencing 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 implemented the role analysis and have used this implementation 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.

***Index Terms*— Program Understanding, Roles, Design Recovery**

I. Introduction

This paper presents a new technique to help developers understand heap referencing properties (such properties capture constraints that involve references between objects in the heap) 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. To the best of our knowledge, the concept that an object's referencing relationships, in conjunction with other properties, should determine its conceptual state was initially developed by Kuncak, Lam, and Rinard [18].

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 is intended to present 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.

A. 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 objects of the same class into different roles. Conceptually, we frame the role separation criteria as a set of predicates that classify objects into roles. Note that this classification of objects into roles can change with time as the objects' referencing relationships change. 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:

- **Referenced-By Relationships:** The functionality of an object often depends on the objects that refer to it. For example, objects of the `PlainSocketImpl` class[1] acquire input and output capabilities when referenced by a `SocketInputStream` or `SocketOutputStream` object. The role separation criteria capture these distinctions by placing objects that are referenced by different fields in different roles. Formally, there is a role separation predicate for each field of each class for a specific number of references. An object `o` satisfies the role separation predicate for the field `f` declared in the class `C` for i number of references if the `f` field in exactly i objects that extend `C` or implement `C`'s interface contains a reference to `o`. The user can specify an upper bound on the number of distinctions to make for a given field. If the upper bound is i, then all objects with at least i such references satisfy the same predicate.
- **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 refers to an actual file descriptor object. To capture these distinctions, our analysis contains role separation criteria that place objects in different roles if they have different non-null

[1]The `PlainSocketImpl` class is the undocumented implementation class for socket communications in the Java API. We used the version of this class included with the Sun JDK version 1.1.8.

fields. Formally, there is a role separation predicate for each non-primitive field of each class. An object `o` satisfies the role separation predicate for the field `f` declared in the class `C` if the object's `f` field is non-null. Although we could, in principle, use more complicated criteria, we have found that the null value criterion usually captures the important properties of the reference-to relation without introducing extraneous distinctions. [2] Formally, there is a predicate for each field of every class. An object satisfies the predicate for the field `f` if the `f` field of the object 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 objects of the `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.

B. 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. The developer specifies a role subspace by choosing an arbitrary subset of the role separation criteria. When operating within a given subspace, the tools coarsen the separation of objects into roles by only keeping the distinctions made by the criteria in that subspace. We envision that developers will 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. These subspaces allow the developer to separate orthogonal concerns into orthogonal subspaces.
- **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.

C. Static vs. Dynamic Analysis

The tool presented in this paper uses a dynamic analysis to extract role information that reflects the observed actions and states from a single execution of the program. Like all dynamic analyses, the extracted information may therefore be incomplete in that different executions of the program may produce different actions and states. Sound static analyses, on the other hand, compute information that reflects the actions and states of all possible executions. The potential drawback is that the information may be more difficult to extract or less precise than the information from a dynamic analysis with similar goals. In particular, we found it easier to build a dynamic tool that extracts role information than to build a static analysis that either verifies or discovers such information [18]. The reasons range from simple engineering issues (we found it is easier to instrument the program and analyze the extracted information than to build a parser and semantic analysis for a complete programming language) to fundamental complexity issues (dynamic analyses only need to deal with the concrete relationships that occur when the

[2] If a developer finds that this criterion does not provide sufficient details about the reference-to relationships, he or she can use the multiple object data structure mechanism described in Section III-B to merge the entire role description of a given class of objects into the roles of the objects that reference them.

program executes, while static analyses need some systematic way to characterize uncertainty in these relationships).

D. Contributions

This paper makes the following contributions:

- **Role-Based Program Understanding:** It introduces the concept that object referencing relationships and method invocation 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 objects 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.

II. 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.

A. 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. This trace consists of a log of the important heap operations, local variable manipulations, and method calls that the program performs. Note that it is not possible to use the standard JVM profile interface [28] to obtain this information — although this interface can generate a notification event for many important operations, it does not generate notification events for object or array field writes. We need this write event information to track the object referencing relationships and synthesize roles from this information.

The tool then reads the trace to extract the role information and convert it into a form suitable for interactive graphical display. The tool evaluates the roles of the objects at method boundaries. We use four abstractions to present the observed role information to the developer: 1) role transition diagrams, which present the observed role transitions for objects 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.

The graphical user interface runs in a web browser with related information linked for easy navigation. We chose this implementation platform because it satisfied all of our user interface needs. The alternative, building our own custom user interface platform, would have substantially increased the engineering effort required to build the system without a corresponding increase in the usability in the system.

B. Role Transition Diagrams

We expect that developers will typically start exploring the behavior of a program by examining role transition diagrams to get a feel for the different roles that objects 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.[3] Note that our tool automatically generates initial names for roles. If the developer is unhappy with the automatically generated names, he or she can rename the roles. The initial names our tool generates consist of a list of the fields that reference objects playing the role, followed by the class name, followed by a list of fields that are non-null in objects playing the role. Special names are generated for the initial role for each class and the garbage role. Excessively complex roles are simply assigned a name consisting of the class followed by a unique number.

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.

[3] 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 [6]. All of the diagrams in this paper were automatically generated using our tool.

Solid edges denote the execution of methods that take JhttpWorker objects as a parameter; dotted edges denote either methods that change the roles of JhttpWorker objects but do not take JhttpWorker objects as a parameter or portions of methods. From the figure we can see that the JhttpWorker.method method transitions JhttpWorker objects from the Initialized JhttpWorker role to the JhttpWorker with filename role. The role JhttpWorker with methodType is an intermediate role that is made visible in the middle of the execution of the JhttpWorker.method when that method made a call to another method. Note that methods may change the roles of objects that are not parameters either indirectly, by changing a heap reference to the object or the object's reachability, or directly, by accessing the object through a global variable or through another object. The diagram always presents the most deeply nested (in the call graph) method responsible for the role change.

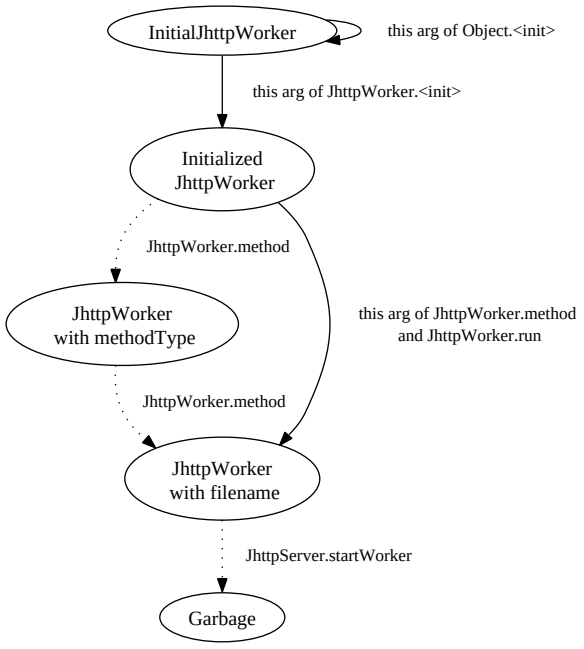

Fig. 1

ROLE TRANSITION DIAGRAM FOR JhttpWorker CLASS

C. 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 objects of the JhttpWorker with filename role have the class JhttpWorker, no references from other objects, no identity relations, and reference objects using the fields httpVersion, fileName, methodType, and client.

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

Fig. 2

SAMPLE ROLE DEFINITION FOR JhttpWorker CLASS

D. 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 structures that the program manipulates. We believe that role relationship diagrams are the primary tool that developers will 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. [4] Figure 4 presents the role relationship diagram that the tool produces when the developer turns this option on. The analysis uses the heuristic that if only one heap reference ever exists to an object, that is likely to be conceptually part of the object that references it. Notice that this heuristic enables the analysis to recognize 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 objects 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 objects of the class have two distinct purposes in the computation: PlainSocketImpl objects that are referenced by Socket objects manage communication over a TCP/IP connection while PlainSocketImpl objects that are referenced by ServerSocket objects accept

[4] Section III-B discusses the specific user-selected policies the analysis uses to discover combinations of objects that conceptually act as a single entity.

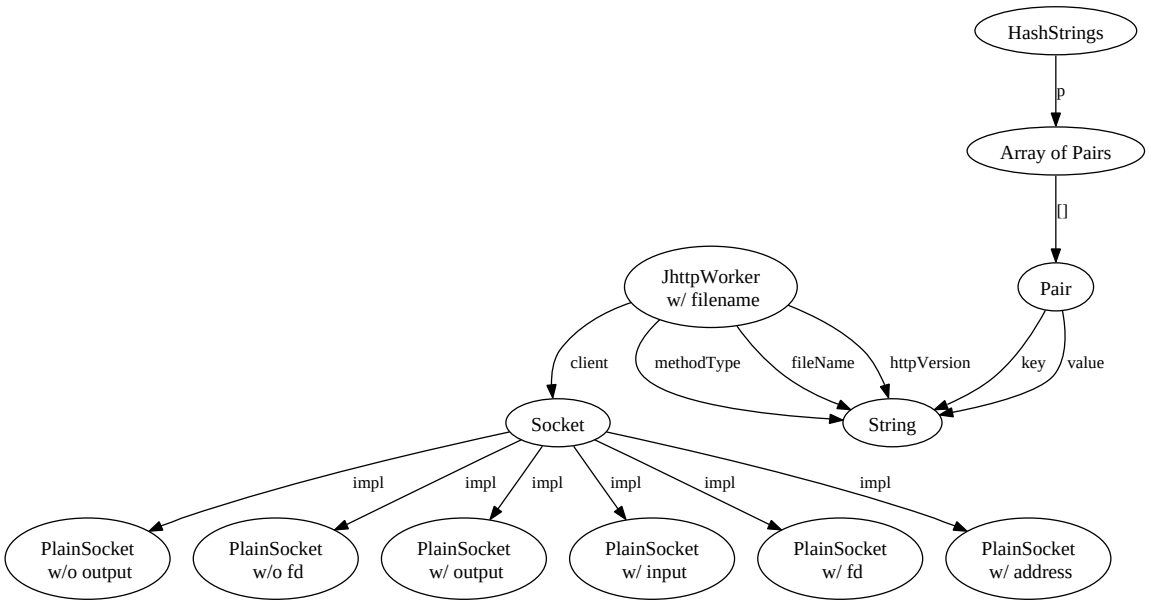

Fig. 3
PORTION OF ROLE RELATIONSHIP DIAGRAM FOR JHTTPSERVER

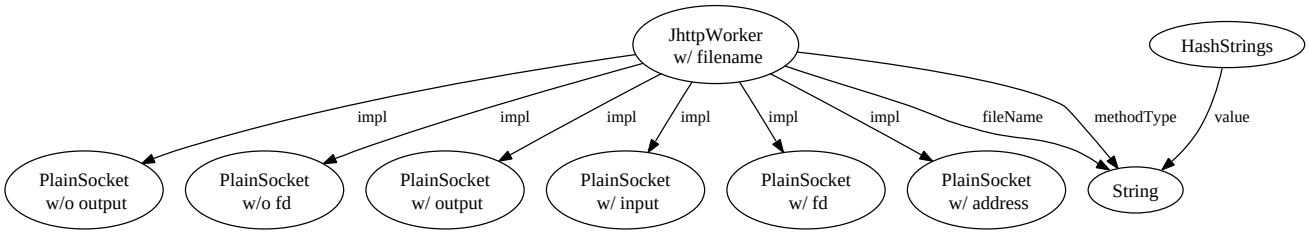

Fig. 4
PORTION OF ROLE RELATIONSHIP DIAGRAM FOR JHTTPSERVER AFTER PART OBJECT ABSTRACTION

new incoming socket connections. We have used the name `ServerPlainSocketImpl` to name the roles that serve the latter purpose. This is an example of a common code reuse pattern in which multiple distinct functionalities are merged into a single object type. In this example, our analysis was able to recover design information about two distinct usage scenarios for the `PlainSocketImpl` class.

Each `PlainSocketImpl` object has a corresponding file descriptor that the underlying operating system uses to implement the socket communication. Even though the state associated with these file descriptors is inaccessible to Java, this state is conceptually part of the object and can affect the object's interface. We note that although the `bind` and `listen` methods do not modify the heap referencing properties of a `PlainSocketImpl` object, they do modify the state associated with the corresponding file descriptor. Moreover, they enable the `accept` method to be invoked on the corresponding `PlainSocketImpl` object. To capture this conceptual change in the object's role, the developer can specify that the invocation of certain methods on an object changes the object's role. Our implementation uses a set of method invocation history predicates to capture these changes. In this example, we configured the tool to include method history predicates for both the `bind` and `listen` methods.

E. 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. The call context section of this interface provides the roles of the parameters at method entry and exit. The write effects section of this interface provides a list of regular expressions summarizing the writes performed by the method. These regular expressions give the path to an object from the parameters of the method and the global variables in terms of the heap at method invocation. The read effects section provides a list of regular expressions summarizing the references that the method reads. The role transitions section provides a list of the role transitions the method has been observed to perform and the corresponding regular expressions specifying the path to the objects that have undergone the role transition. The presence of the keyword `NEW` for a regular expression indicates that the object was allocated within the scope of the method. 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 of Initial InputStream and PlainSocket w/fd. When it executes, it changes the roles of

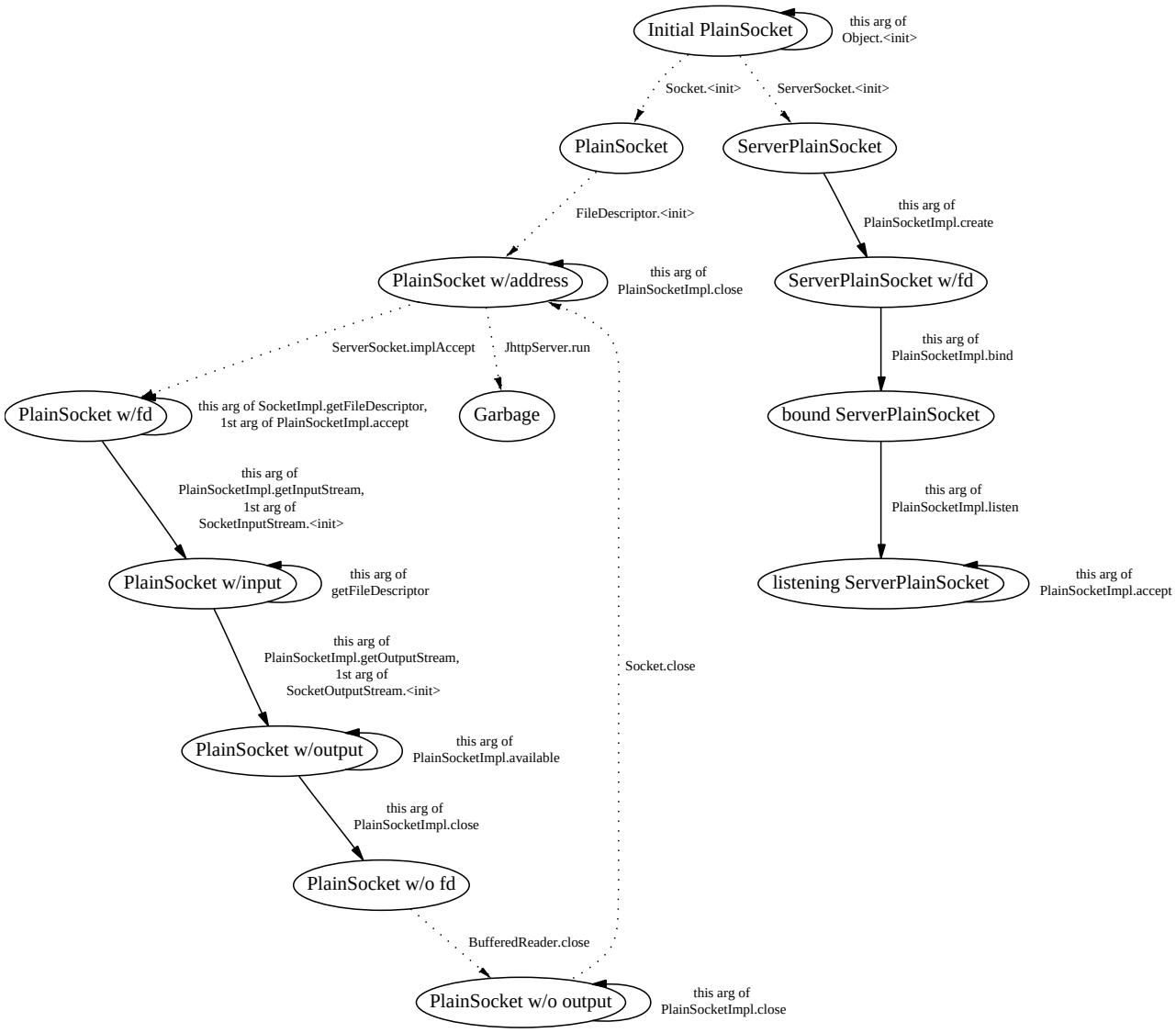

Fig. 5
ROLE TRANSITION DIAGRAM FOR THE `PlainSocketImpl` CLASS

these objects to InputStream w/impl and PlainSocket w/input, respectively.

Enhanced method interfaces provide the developer with additional information about the (otherwise implicit) assumptions 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 insight for its successful use in other contexts. [5]

F. 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. [6]

[5]For example, the method may require that one of its parameters play a specific role. The enhanced method would reveal this to the developer.

[6]Section VI presents some examples of the tool being used to discover the following properties for a set of benchmarks.

- **Consistency Properties:** Our analysis can discover program-level data structure consistency properties. For example, our analysis may discover that while an object may participate in several data structures over its lifetime in the program, at any given time, it participates in at most one of those data structures. The analysis can also discover fine-grained properties within an individual data structure, for example that the `next` and `prev` references in a doubly-linked list are inverses. Section VI provides several examples of additional data structure properties that occur in our set of benchmark applications.
- **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.

```
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
```

Fig. 6
ENHANCED METHOD INTERFACE FOR `SocketInputStream` INITIALIZER

- **Multiple Uses:** Code factoring minimizes code duplication by producing general-purpose classes that can be used in a variety of contexts. In some cases, a single class can provide different functionality, but due to code duplication these different pieces of functionality may be merged into a single class. But this practice obscures the different purposes that different objects 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.

III. DYNAMIC ANALYSIS

We implemented the dynamic analysis as several components. The first component uses the MIT FLEX compiler [7] to instrument Java programs to generate execution traces. Because our compiler accepts Java byte codes as input and generates native code as output, it does not require source code. The instrumented program assigns unique identifiers to every object and records relevant heap and pointer operations in the execution trace. Relevant operations in this case include writing pointer values to arrays or fields, cloning objects, creating new objects, reading pointer values, creating or changing local variable references to objects, and method calls and returns. The second component uses this trace to replay the program's manipulation of 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.

A. Predicate Evaluation

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

[7]Available at `http://flex-compiler.csail.mit.edu/`.

- **Referenced-By:** 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 referenced-by 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. These predicates are designed to identify pairs of fields with an inverse relation — these are commonly used to provide the ability to traverse a data structure both forwards and backwards.
- **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. We discuss the reachability algorithm in greater detail in Section III-D.

  Programs often use temporary variables to traverse or manipulate a data structure. Using temporary variables to separate objects into different roles would have the effect of separating objects into different roles with meaningless distinctions. Such roles would likely make navigating the generated role abstractions more difficult. To avoid undesirable separations caused by temporary variables, we developed two rules to identify variables that are the roots of data structures. We believe that the conceptually important references to data structures are likely to be older. Furthermore, the entire data structure is likely to be reachable from these conceptually important references. We designed the following two rules to eliminate references which are not likely to be conceptually important: 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.

In theory, these rules can fail to extract the conceptual root reference of a data structure for cyclic data structures if some extraneous reference to the data structure was created before the root reference, or if some other data structure references the same object that the conceptual root variable references. Furthermore, these rules can lead to extraneous roots if a temporary reference exists to the root object. In practice, we believe that these rules will rarely miss including a reference from root variable and will not include many extraneous root variables.

- **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 on every object whose role may have changed since the last method entry or exit point. Whenever an object is observed to transition from one role to another, the role change is recorded along with the method that performed the change. This role transition information is used to construct the role transition diagram and is also presented in the enhanced method interfaces. Furthermore, the roles of the objects that reference or are referenced by the object are recorded whenever a new reference is created or an object changes roles. This referencing information between roles is used to construct the role relationship diagram. When a method performs a write, changes an object's role, or obtains a reference to an object this information is recorded for use in the enhanced method interface.

We allow the developer to coarsen the granularity of role evaluation by declaring methods *atomic*, in which case the analysis attributes all role transitions that occur inside the method to the method itself. When a method is declared *atomic*, the analysis does not compute the roles of objects for methods that the *atomic* method (transitively) invokes. 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 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.

B. 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. For example, the developer may be attempting to understand the use of a particular class and desire to see role information for just that class. Or the developer may be interested in understanding how an object of a particular class interacts with objects it references, and may like to see role information for the combination of multiple objects. On a per class basis, the developer can specify whether to merge one object's role into another object's role. Furthermore, the developer can specify a default policy for the classes for which the developer does not explicitly specify a policy. This default policy allows the developer to only specify the policies for the classes in which the developer is currently interested. This approach is especially important for large programs, given the potential developer overhead associated with explicitly specifying a policy for every class.

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 at most one heap reference at a time exists to the object. The analysis implements these policies by examining the execution trace that the instrumented application generates. Note that the merge policies are based on properties that depend, in general, on the entire trace: any partial examination of the trace will, in general, be unable to determine that the unexamined part of the trace does not create multiple or multiple simultaneous references to a given object.

The analysis therefore uses a two-pass approach to merge multiple objects into a single entity. The first pass applies the merge policies to determine which objects to merge into the objects that refer to them; the second pass uses the list of merged objects from the first pass to appropriately assign the roles for merged objects.

C. Method Effect Inference

For each method execution, the dynamic analysis records the reads, writes, and role transitions that the execution performs. There is a method effect summary for each combination of a method and an assignment of roles to the parameters of the method. The analysis combines the results of all invocations of a method with the same assignment of roles to the parameters. 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 method 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. The path table includes not only objects that the method may read or write, but also any objects that the method may cause to transition to a different role. Since reachability changes may change the role assignment of an object, the table must include a concrete path to any objects whose reachability information may change. 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. [8] If the execution creates a new object, the tool adds a special NEW token to the path table; this token represents the path to that object.

The tool obtains the regular expressions in the method effect summary by applying a set of rewrite rules to the extracted concrete paths. These concrete paths consist of a starting point, a parameter of the method or a global variable, and a list of fields that give the concrete path to the object in terms of the heap at invocation time. 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 contains the concrete path and ϵ represents an empty concrete path or regular expression. 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. In addition to these rules, our tool uses a set of rules to determine whether two regular expressions can be merged. If the regular expressions of two of the same effects can be merged, the effects are merged.

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.

For example, consider the concrete path $f.g.f.g.f.g$ where field f is declared in the type G and references an object of type F and fields g and h are declared in the type F and references an object of type G. The initial state for the rewrite algorithm is $\langle \epsilon, f.g.f.g.f.h \rangle$. The algorithm begins by applying rule 7 to this state four times to generate the state $\langle f.g.f.g, f.h \rangle$. The algorithm next applies rule 5 to generate the state $\langle (f.g)^*, f.h \rangle$. The algorithm then applies rule 7 two more times to generate the state $\langle (f.g)^*.f.h, \epsilon \rangle$. Finally, the algorithm applies rule 6 to generate the final state $\langle (f.(g \mid h))^*, \epsilon \rangle$.

[8] The tool calculates concrete paths by tracking how the method obtains a reference to the objects in the heap. Whenever the method first obtains a reference to an object, the tool records, in a table, the field and the object that the method dereferenced. If a change to the heap affects an object's reachability, the tool records the path information for that object when the tool updates the reachability information for that object. The tool uses this table to efficiently generate paths from the parameters of the method and the global variables to the objects. These paths refer to version of the heap that existed when the method was invoked.

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

Fig. 7
REWRITE RULES FOR PATHS

When a method performs a read operation on an object or causes the role an object plays to change, the analysis records the change as a read effect or a role transition effect. The analysis also records an expression that identifies the objects involved in the operation in terms of a path through the heap. The expression gives the starting point of the path (either a parameter of the method or a global variable) and a regular expression that summarizes the sequence of fields in the path. When a method performs a write, the analysis records a write effect and similar path information as for a read effect, specifically, the field the method wrote and the path expressions for both object containing the field and the object reference written to the field.

Finally, the inference algorithm must also recognize object creations and writes of null references to object fields. We use the NEW token to denote objects created during the method's invocation. We use the NULL token to denote writing a null reference to an object's field.

D. Incremental Reachability Algorithm

Our results indicate that most methods make relatively small changes to the heap. An incremental approach to computing reachability should therefore be more efficient than completely recomputing the reachability information whenever it is needed. Our tool records a list of changes performed to the heap since the last reachability computation.

When computing reachability, the tool starts by processing the list of removed references in the heap. For each removed reference, it marks the destination object as possibly unreachable from the set of local and global variables that both the destination and origin objects were previously reachable from. This set contains the root variables from which the destination of the removed reference may no longer be reachable. After processing the list of removed references, the tool propagates

the possibly unreachable sets of roots through the heap in the direction of the references in the heap. During this propagation step, the tool checks that the propagation step never marks an object directly referenced by a local variable or global variable as possibly unreachable from the same local or global variable.

The tool next adds any new references from local or global variables to the reachability sets of the objects that they reference. Finally, a work set algorithm propagates reachability information through the heap. The work set initially includes any objects that reference the previous destinations of any removed references and any objects that are the sources of any new references. The work set algorithm then propagates the reachability information through the heap.

Formally, we represent the heap as a set of objects O, a set of heap reference edges $E \subseteq O \times F \times O$ where F is the set of fields and array indices, and a set of variable references $L \subseteq V \times O$ where V is the set of local and global variables. We represent the reachability information using the reachability set $R \subseteq O \times V$ that maps an object to the set of variables from which the object can be reached. We define $R(o) = \{v \mid \langle v, o\rangle \in R\}$.

The incremental analysis takes as input a set of removed heap references $E_R \subseteq O \times F \times O$; a set of removed variable references $L_R \subseteq V \times O$; a set of newly created heap references $E_{New} \subseteq O \times F \times O$; a set of newly created variable references $L_{New} \subseteq V \times O$; a tuple containing the set of objects, the current heap references, and the current variable references $\langle O, E, L\rangle$; and the reachability set R.

Figure 8 presents the incremental reachability algorithm. The incremental reachability algorithm internally uses the work set $S \subseteq O$ to store the objects whose reachability information may need to be propagated. The algorithm begins by initializing this set to the empty set. The algorithm next calls the ProcessRemovedReferences procedure. The ProcessRemovedReferences procedure internally uses a workset $K \subseteq O \times V$ of tuples comprised of an object and a label to maintain a list of variables that an object may no longer be reachable from. We define $K(o) = \{v \mid \langle o, v\rangle \in K\}$. The procedure initializes the set K in lines 2-5 using the set of heap references and variable references that have been removed since the last reachability computation. The algorithm next loops through each object o that serves as a key in K. The algorithm then adds any objects that reference o to the set S of objects whose reachability information may need to be propagated. The algorithm then looks up in K the variables from which the object o may no longer be reachable. The algorithm next loops through all of the objects that are referenced by the object o and propagates this list of variables from which the object may possibly be unreachable.

The incremental reachability algorithm next calls the ProcessNewReferences procedure. This procedure propagates reachability information for both new references and any references that were mistakenly removed in the previous procedure. Lines 1-4 process any newly created references from variables. These lines add the destination of these references to the workset S and the new reference to reachability information set R. Lines 5-6 process the newly created heap references in a similar manner. Finally, the algorithm loops through the

```
IncrementalReachability(E_R, L_R, E_New, L_New, R, E)
1 S := {}
2 ProcessRemovedReferences(E_R, L_R, R, E, S)
3 ProcessNewReferences(E_New, L_New, R, E, S)

ProcessRemovedReferences(E_R, L_R, R, E, S)
1 K := {}
2 for each tuple ⟨o_src, f, o_dst⟩ in E_R do
3     K := K ∪ ({o_dst} × R(o_src))
4 for each tuple ⟨v, o⟩ in L_R do
5     K := K ∪ ⟨o, v⟩
6 while K is not empty
7     pick a key o in K
8     S := S ∪ {o' | ∃f ∈ F, ⟨o', f, o⟩ ∈ E}
9     S_R := K(o), K := K − ({o} × S_R)
10    R := R − ({o} × (S_R − {v | {v, o} ∈ L}))
11    for each o_dst in {o' | ∃f ∈ F, ⟨o, f, o'⟩ ∈ E}
12        if R(o_dst) ∩ S_R ≠ {}
13            K := K ∪ ({o_dst} × R(o_dst))

ProcessNewReferences(E_New, L_New, R, E, S)
1 for each tuple ⟨v, o⟩ in L_New
2     if ⟨o, v⟩ is not in R
3         R := R ∪ ⟨o, v⟩
4         S := S ∪ {o}
5 for each tuple ⟨o, f, o'⟩ in E_New
6     PropagateReachability(o, o', R, S)
7 while S is not empty, select o from S
8     remove object o from S
9     for each o_dst in {o' | ∃f ∈ F, ⟨o, f, o'⟩ ∈ E}
10        PropagateReachability(o, o_dst, R, S)

PropagateReachability(o, o', R, S)
1 if R(o) ⊈ R(o')
2    R := R ∪ (⟨o'⟩ × R(o))
3    S := S ∪ {o'}
```

Fig. 8
INCREMENTAL REACHABILITY ALGORITHM

objects in set S to propagate their reachability information to any objects they reference, and then adds these referenced objects to S.

The overhead of incremental algorithm is determined by the number of heap references through which the algorithm must propagate the possibly unreachable set of variables and the number of heap references through which the algorithm must propagate the reachability information. In the worst case, the incremental algorithm does not perform better than a standard reachability algorithm — it can take time proportional to the number of references in the heap times the number of global and local variables that reference the heap. In practice, we expect that many methods will make changes that require propagating reachability information through only a small part of the program's heap and therefore that the algorithm will perform much better than the worst case bound.

E. Multiple Executions

Our tool supports the analysis of traces from multiple executions. We have architected our multiple trace support as follows: the traces are processed individually by the analysis and then the web frontend merges the analysis results for the individual traces into a single merged result. The benefit of this approach is that it enables our implementation to parallelize the analysis of the traces. The basic approach is straightforward. Since the role transition diagrams capture the range of possible behaviors of an object, the combined role transition

diagram is simply the union of the role transition diagrams for the individual traces. Similarly, the combine role relationship diagram is simply the union of the role relationship diagrams for the individual traces. While the currently implemented web interface does not currently process the enhanced method interfaces, it is conceptually straightforward to combine enhanced method interfaces from different traces. The algorithm would simply take the union of the enhanced method interfaces from the individual traces. If the same enhanced method interface appears in multiple traces, the algorithm would take the union of the read and write effects from different instances of the same enhanced method interface.

F. Role Subspaces

Our tool allows the developer to define multiple role subspaces and to 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.

IV. User Interface

The user interface presents four kinds of web pages[9]: 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.

[9]We chose a web interface because it provides a convenient, reliable, cross platform mechanism for communicating the results of the analysis. The web interface allows us to easily support viewing multiple pages at once, linking nodes in graphs to descriptions of the underlying roles, and would enable us to easily link analysis results to online resources.

The user interface allows the developer to create and manipulate multiple role subspaces. The developer can create a new role subspace by selecting the set of predicates to capture the desired role separation criteria. The developer can then define a view, which allows the developer to see the role transition diagrams, the role relationship diagrams, and the enhanced method interfaces generated using one or more role subspaces. 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 set of roles appearing in diagrams isomorphic[10] 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.

V. 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 objects of the class in the computation.

During this activity, we were interested in obtaining a broad overview of the actions of the program. [11] 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

[10]There exists a one-to-one and onto mapping between the roles appearing in a view with multiple subspaces and the roles appearing in a subspace that has the union of the role separation criteria from all the subspaces as its role separation criteria.

[11]We expect that many developers will be interested in understanding or debugging a particular aspect of the program. We believe that our tool will be useful for this purpose. In fact, we believe that much of the information we obtained when obtaining a broad overview would useful for developers debugging or modifying very specific pieces of the applications.

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 believe that maintainers will also find it useful.

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. This multiple level approach to program understanding is well known; developers often use bottom-up [23] and top-down [2] approaches for understanding software. Given our experience using this feature of our role analysis tool, we believe that this ability will prove valuable for any program understanding tool.

VI. 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 SPECjvm98 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 JOlden benchmark suite [12]; and Em3d, a scientific computation in the JOlden benchmark suite.

A. 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. To generate a trace to analyze, we simply selected one of the Jess example problems included with the Jess distribution and ran the instrumented version of Jess on that problem to produce the trace. We analyzed the trace for Jess with our tool, and then investigated the role transition diagrams for the classes. From the quick overview of role transition diagrams, it appeared to us that the Node structures used by Jess would be the most interesting to a developer.

Consider, for example, objects of the `Node1TELN` class. Figure 9 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 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. For this benchmark, we used two different test cases to generate two different execution traces. We discovered in the second execution trace that Jess can create node objects, but not freeze them. We used the multiple execution functionality of our tool to combine the traces from the two executions to generate Figure 9. Due to the fact that the names of the fields in the program were informative, the automatically generated role names were very helpful. Only minimal renaming was done for the purpose of aesthetics.

The generated extended method interfaces provide information about the assumptions that several key methods make about the roles of their parameters. Specifically, they 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. This invocation sequence constraint could also be determined using specification mining techniques[1].

The role definitions also provide information about the network's structure, specifically that all of the nodes in the network have either one or two incoming edges. Each fully constructed object 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 object of the `Node2` class, on the other hand, has exactly two references from `Successor` objects, indicating that `Node2` nodes have exactly two incoming edges.

B. Direct-To

Direct-To is a prototype Java implementation of a component of the Center-Tracon Automation System (CTAS) [13]. 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 ran Direct-To on a short input file consist of a few aircraft to generate a trace. We looked at the role transition diagrams for the different classes and identified the `Flight` class as a central class in the computation.

We first discuss our experience with the `Flight` class, which represents flights in progress. Figure 10 presents the role transition diagram for the `Flight` class. 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 examination of

[12] Available at `ftp://ftp.cs.umass.edu/pub/osl/benchmarks/jolden.tar.gz`.

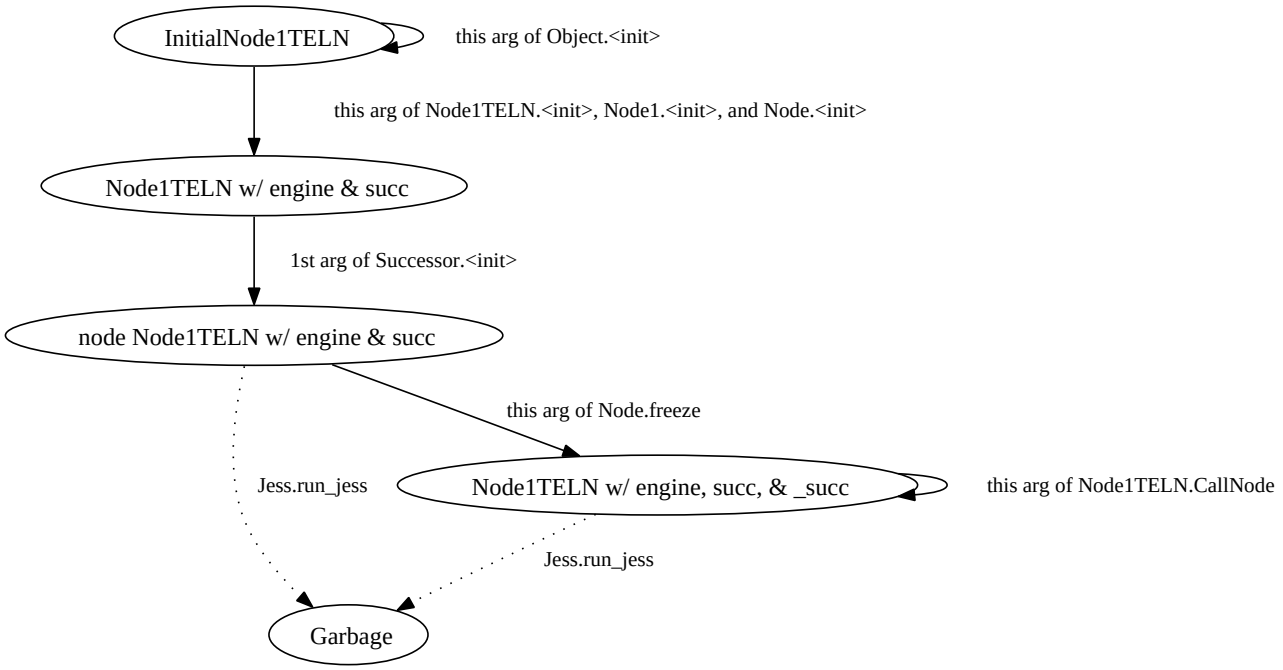

Fig. 9
ROLE TRANSITION DIAGRAM FOR THE `Node1TELN` CLASS

the role transition diagram reveals that 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. Due to the fact that the initialization of `Flight` class objects is performed by the combined actions of several methods from different classes, discovering the initialization sequence by simply examining the code is not straightforward.

Roles also enabled us to untangle the different ways in which the program uses objects of the `Point4d` class. Specifically, the program uses objects of this class to represent aircraft tracks, trajectories, and velocities. This distinction is useful, because the operations that are valid on `Point4d` object used in a velocity are different from the operations that are valid on a `Point4d` object used by a trajectory. For example, multiplying a `Point4d` object used as a velocity by time is a legal operation while the same operation performed on a `Point4d` object used as a position in a trajectory is nonsensical. 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 likely uses the corresponding objects for disjoint purposes.

C. 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. To generate a trace for Tagger, we simply ran the instrument version on the example file included with Tagger.

It turns out that there are dependencies 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 output suppression with a matched pair of actions: a suppress action and a corresponding unsuppress action. Figures 11 and 12 give the role transition diagrams for the unsuppress action and the suppress action, respectively. 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. This is visible in role transition diagram for the unsuppress action class — the unsuppress object transitions from the unbound role (`STANDARD unsuppress_action StandardEngine$16 w/ generator & this`) to the

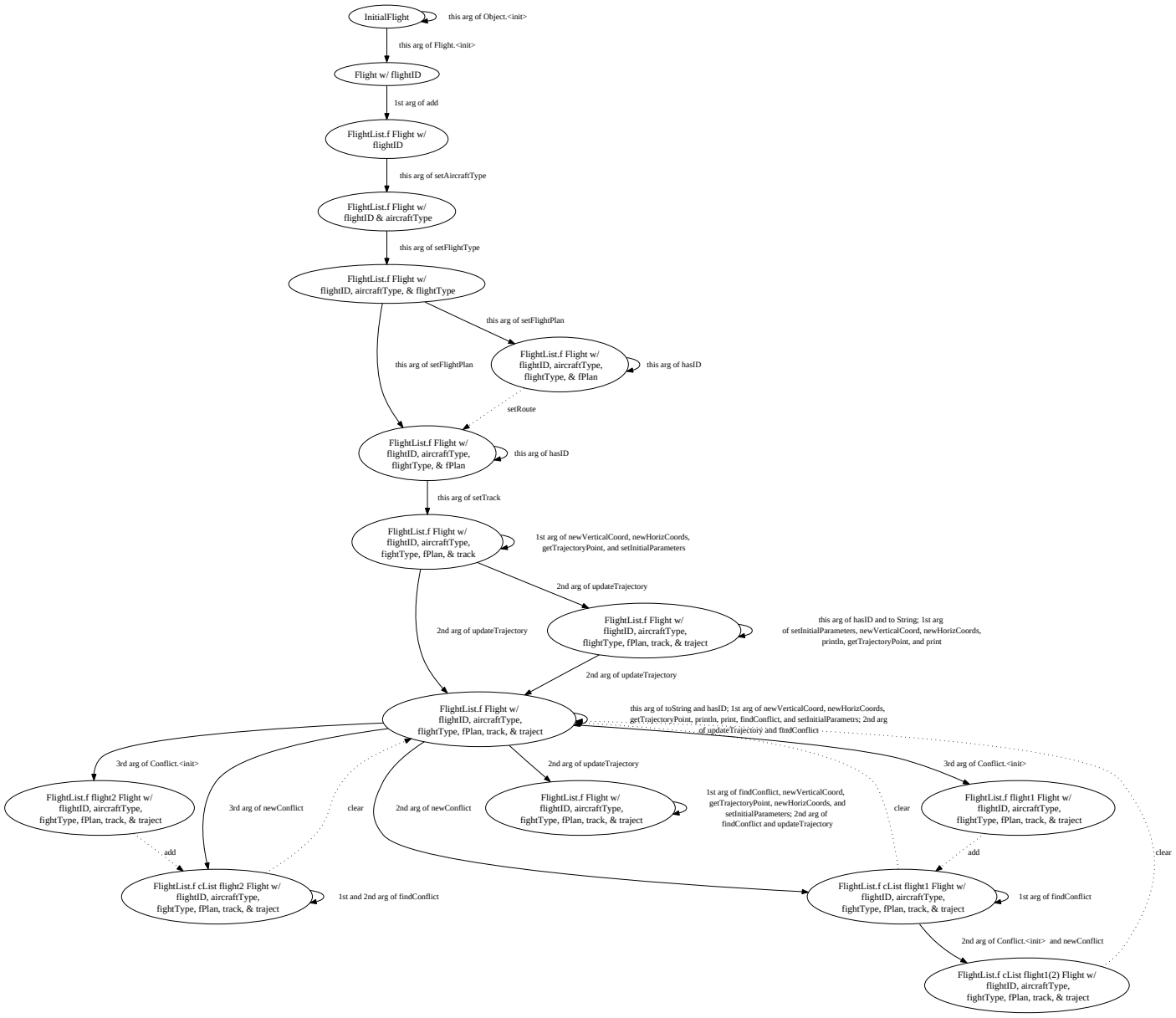

Fig. 10
ROLE TRANSITION DIAGRAM FOR THE `Flight` CLASS

bound role (`STANDARD element unsuppress_action StandardEngine$16 w/ generator & this`) when the suppress action is performed and transitions back to the unbound role when the unsuppress action is performed. We were also able to observe a role-dependent interface [13] — the method that executes actions always executes actions that are bound to tokens.

D. 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. To generate a trace file for Treeadd, we simple ran the Treeadd benchmark.

Figure 14 presents the region of the role relationship diagram that contains the roles of `TreeNode` objects. By examining this diagram, enhanced method interfaces, and the linked role definitions, we were able to determine that the structure returned by the tree construction method did in fact comprise a tree — the tree construction method returns a root `TreeNode` object playing the role `TreeNode w/ right & left`, which according to the role definition has no references from `left` or `right` fields of other `TreeNode` objects. The other `TreeNode` roles have exactly one reference from the `left` or `right` field of another `TreeNode`. Combining these two pieces of information, allows us to infer that the structure returned by the tree construction method is a tree.

[13] A role-dependent interface is a method that expects its parameters to have a certain role.

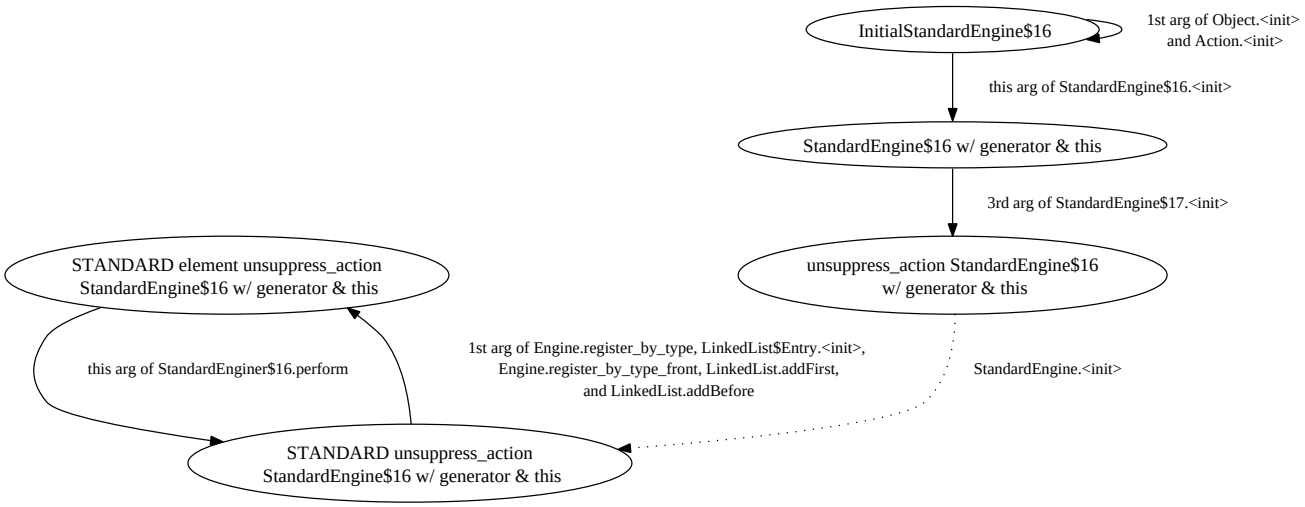

Fig. 11

ROLE TRANSITION DIAGRAM FOR THE UNSUPPRESS ACTION CLASS

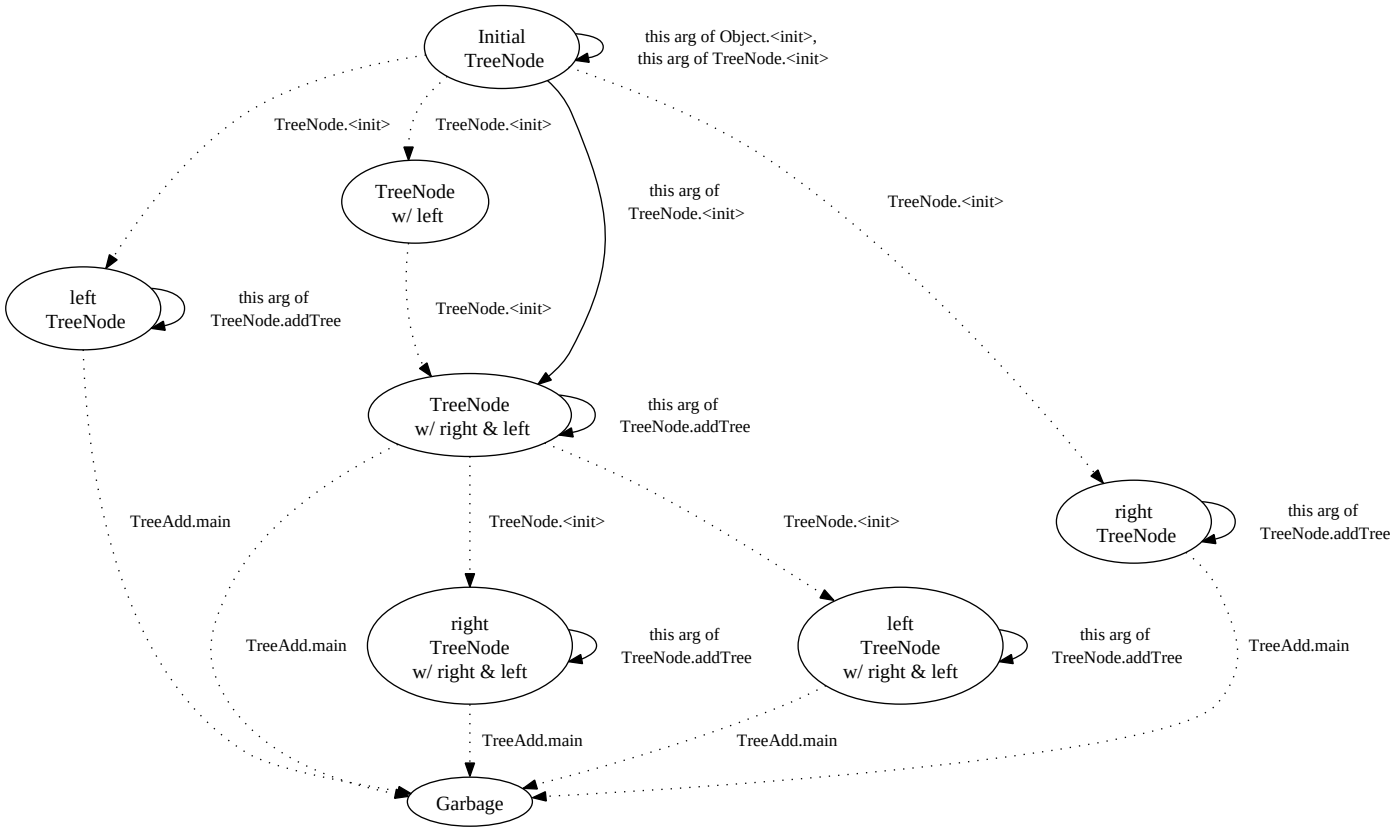

Fig. 13

ROLE TRANSITION DIAGRAM FOR THE `TreeNode` CLASS

Figure 13 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.

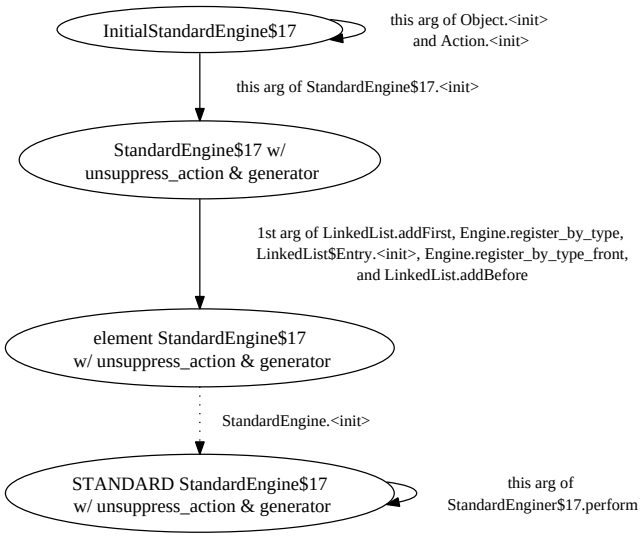

Fig. 12
ROLE TRANSITION DIAGRAM FOR THE SUPPRESS ACTION CLASS

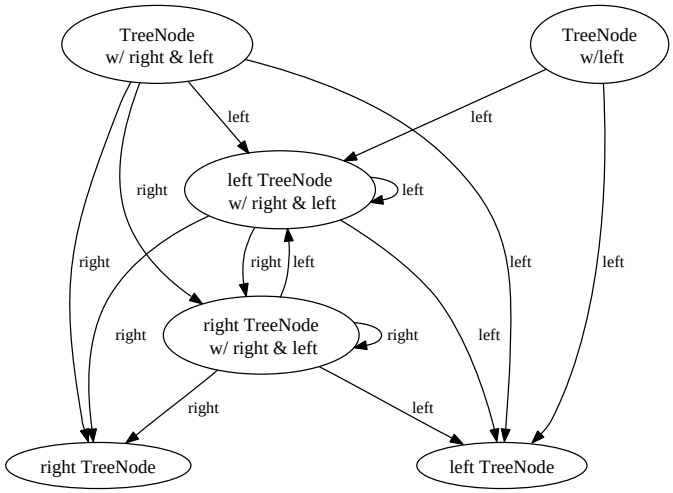

Fig. 14
ROLE RELATIONSHIP DIAGRAM FOR THE `TreeNode` CLASS

E. 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. Figure 15 gives the role transition diagram for the `Node1Enumerate` class. 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.

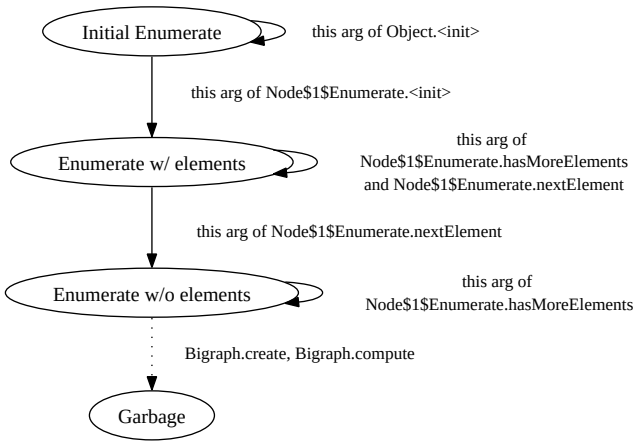

Fig. 15
ROLE TRANSITION DIAGRAM FOR THE `Node.Enumerate` CLASS

F. 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 objects of a single class for several different purposes in the computation. In all of these cases, the role analysis enabled us in a matter of minutes to 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, we believe that understanding heap allocated data structures is key to understanding the program. Roles help

developers discover potential 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:** We expect that developers could 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. The enhanced method interfaces and role transition diagrams can also help the developer quickly isolate the bug.
- **Documentation:** Developers often need to document high-level properties of the program. We believe that roles may provide an effective documentation mechanism, because they come with a set 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. For instance, use cases can be thought of as executing a state transition on the objects involved to update information in the system. UML class diagrams and state charts are more obvious instantiations of such design formalisms. 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.

G. Analysis Overhead

We measured the instrumentation and analysis overheads for our tool on the benchmark applications. We ran each on benchmark on a 2.2 GHz Core 2 Duo with 1 GB of RAM. Table I presents the overhead measurements for our benchmarks. For each benchmark, we report: (1) the time for the normal, un-instrumented version to execute, (2) the time for the instrumented version to execute and store the trace, (3) the time for our analysis to process the trace, and (4) the size of the trace. We compute that the slowdown due to instrumentation ranges from a low of 16x for the Direct-To benchmark to a high of 122x for Jess benchmark with an average of 51.6x. Note that because the instrumentation primarily records changes to heap referencing properties, the instrumentation overhead varies depending on how much data structure manipulation the application performs.

Bench-mark	Normal Execution	Instrumented Execution	Analysis Time	Trace Size
Jess	0.07s	8.6s	20m32s	286.6 MB
Direct-To	0.59s	9.5s	3m3s	292.7MB
Tagger	.08s	4.9s	5m13s	153.3MB
Treeadd	.003s	.059s	0.38s	1.8MB
Em3d	.006s	.23s	60s	7.3MB

TABLE I
OVERHEAD MEASUREMENTS

We next compute the time taken to analyze the trace in seconds per megabyte of trace file. We found that Em3d took the most time per megabyte with a time of 8.2 seconds/MB. We found that Treeadd took the least time per megabyte with a time of 0.211 seconds/MB. We noted that Jess and Direct-To have similar sized trace files, but significantly different analysis times. We profiled the role inference analysis on these two benchmark applications to help understand the difference in analysis times. We found that computing the reachability information took 55.5 % of the analysis time for the Jess benchmark while it only took 26.6 % of the analysis time for the Direct-To benchmark.

The reachability computations dominate the worst-case analysis time. In the worst case, the incremental reachability algorithm can take time proportional to the number of edges in the heap times the number of possible root variables. Since the tool recomputes reachability at every method boundary, the worst case analysis time is proportional to the worst case reachability bound times the number of method calls.

In practice, we believe that the incremental reachability algorithm will yield significant performance improvements for many benchmarks as it will only recompute reachability for small portions of the heap. In this case, the analysis time should be approximately proportional to the execution time of the program.

The multiple object data structure functionality can theoretically affect the complexity of the analysis. If the developer were to use the always merge policy for classes that are instantiated many times to build large linked data structures, the analysis could potentially have to propagate a role change for an object through many other objects. In practice this is unlikely to pose a problem as this would generate a huge of number of roles for such classes and the resulting graphs would be very difficult to interpret.

VII. 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 analysis for automatically discovering or verifying properties of linked data structures.

A. Design Formalisms

Early design formalisms identified changes in abstract object or component states as an important aspect of the design

of the program [26]. 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 [25] and Alloy [12] 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 [11], and researchers have proposed several methodologies for realizing these roles in the program [11], [10], [15]. 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.

B. Program Understanding Tools

Daikon [8] 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 [9]. 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. Furthermore, our tool can discover and communicate the changes that occur to an object's state during a program's execution. As a result, our tool allows the developer to discover a rich new class of program invariants. This class of program invariants relate an object's state and changes to the object's state to the program's actions.

Jinsight [4] also extracts information from a program's execution. Jinsight allows the developer to see a histogram view, which shows a program's use of space and memory on a class by class basis. It also provides two different views showing method invocation information. Jinsight also provides pattern views for references and invocations of the execution. Jinsight appears to be most useful for understanding how a program uses computational resources and identifying opportunities for optimization. Furthermore, Jinsight may be useful for identifying coding errors that leak computational resources. Our tool is more useful for understanding deeper properties of the objects in a program, and how these objects interact with the program's code. We believe that our tool is useful for finding subtle bugs in the use of objects.

Womble [14] and Chava [16] 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. These tools statically group all objects 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. Since our role analysis tool utilizes an abstraction that can capture an object's state changes, our tool can discover and communicate program properties that depend on the state of the object.

Lackwit [22] uses a static type inference algorithm to automatically refine the C type system. This refinement allows abstract datatypes to be separated at a finer granularity than the standard C type system provides. Various extensions allow the system to find memory locations that are statically guaranteed to not be written to, read from, or freed. However, these extensions cannot detect objects that might not be written to, read from, or freed. Although Lackwit statically refines the C type system, it does not capture the changes an object undergoes during a program's execution. Therefore, it cannot discover properties that relate an object's state to the actions of the program.

Ajax [21] uses a static alias analysis to automatically generate object models. The tool uses alias information to generate object models that refine what can be generated from the Java type system alone. The object models Ajax generates are similar to our role relationship diagrams, but our tool provides the capability to represent multiple object data structures as a single conceptual entity. Furthermore, our tool allows an object's role (and thus the set the object is associated with in an object model) to change during the execution of the program.

Engler et al. [7] developed a technique that uses patterns to infer specifications for programs. These patterns can infer temporal rules such as lock x is acquired before accessing b and no reads after a free. They implemented this technique and used it to find bugs in Linux kernel. Kremenek et al. [17] generalizes this approach to infer returns ownership or claims ownership annotations for functions. Li and Zhou [19] generalize the technique of Engler et al. to support a wider variety of programming rules and to eliminate the need for templates that describe the possible forms of rules. Their technique is based on the frequent itemset data mining technique. The CHRONICLER tool extends this work with path sensitivity to discover condition checks [24]. Our work differs in that is able to discover more complex ordering interface constraints in the form of finite state machines and that it is able to discover relations between an object's interface and the object's heap reference properties.

Shoham et al. [27] use an abstract trace collection technique to build finite state machines of events for an object. Nanda et

al. [20] present a static analysis that infers typestate properties. Their approach does not require test cases, but only scales to small examples. They use an emulation technique to emulate client actions on data structure components. We expect that these techniques complement our technique: these techniques guarantee that their generated finite state machines contain a conservative approximation of all legal transitions while our technique guarantees that our generated finite state machines only contain legal transitions. We expect that depending the task at hand, software developers need to know both types of information.

Ammons et al. [1] present a dynamic analysis that monitors function calls to a component to automatically extract finite state machines that describe the component interface. Their technique requires the user to write function attributes that describe whether a function defines or uses an abstract object value. Our work differs in that it relates heap reference properties to an object's interface.

Wasylkowksi et al. [29] presents a static analysis that can extract and check finite state automata for method calls on individual objects.

Whaley et al. [30] use a combination of static analysis and dynamic analysis to discover finite state machine component interface models. Their approach uses static analysis to identify side effect free methods and to identify possible illegal transitions.

Cook and Wolf [3] explore the problem of generating finite state machines for traces of uninterpretted events.

A previous version of this paper appeared in ICSE 2002 [5]. This version of the publication adds more details on the generation of role names, method effect inference, the incremental reachability algorithm, measurements of the analysis overhead, and our experiences using the tool. Furthermore, we have extended the tool to support the analysis of multiple execution traces from a single program.

C. Verifying Data Structure Properties

The analysis presented in this paper extracts role information for a set of execution traces of the program. 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 [18]. 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.

VIII. 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, potential 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.

IX. 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. An earlier version of the article appeared in the Conference Record of the 24th International Conference on Software Engineering.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM Symposium on the Principles of Programming Languages*, 2002.

[2] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[3] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engingeering and Methodology*, 7(3):215–249, 1998.

[4] W. De Pauw, G. Sevitsky, and E. Jensen. Jinsight: A tool for visualizing the execution of java programs. http://www-106.ibm.com/developerworks/java/library/j-jinsight/.

[5] B. Demsky and M. Rinard. Role-based exploration of object-oriented programs. In *ICSE02*, May 2002.

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

[7] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[8] 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.

[9] 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.

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

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

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

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

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

[15] B. Jacobs. Patterns using Procedural/Relational Paradigm. `http://www.geocities.com/tablizer/prpats.htm`.

[16] 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.

[17] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, 2006.

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

[19] Z. Li and Y. Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[20] M. G. Nanda, C. Grothoff, and S. Chandra. Deriving object typestates in the presence of inter-object references. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object Priented Programming, Systems, Languages, and Applications*, 2005.

[21] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, "Carnegie Mellon School of Computer Science", 2002.

[22] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *International Conference on Software Engineering*, pages 338–348, 1997.

[23] N. Pennington. Comprehension strategies in programming. In *Empirical Studies of Programmers: Second Workshop*, pages 100–112, 1987.

[24] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th International Conference on Software Engineering*, pages 240–250, 2007.

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

[26] 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.

[27] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[28] Sun Microsystems. Java Virtual Machine Profiler Interface (JVMPI). `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/`.

[29] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44, New York, NY, USA, 2007. ACM.

[30] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2002.