Static and Dynamic Analysis of Call Chains in Java

Atanas Rountev Scott Kagan Michael Gibas
Department of Computer Science and Engineering
Ohio State University

{rountev,kagan,gibas}@cis.ohio-state.edu

ABSTRACT

This work presents a parameterized framework for static and dynamic analysis of call chains in Java components. Such analyses have a wide range of uses in tools for software understanding and testing. We also describe a test coverage tool built with these analyses and the use of the tool on a real-world test suite. Our experiments evaluate the exact precision of several instances of the framework and provide a novel approach for estimating the limits of class analysis technology for computing precise call chains.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Measurement, Algorithms

Keywords

Static analysis, dynamic analysis, call graph, call chains

1. INTRODUCTION

A *call (multi)graph* is a widely used representation of calling relationships among methods. For example, an edge from method m_i to method m_j may represent the fact that some call site inside m_i potentially invokes m_j. In object-oriented software, call graph construction requires taking into account the effects of virtual dispatch. To achieve this, some form of *class analysis* is necessary to determine the classes of the objects to which reference variables may point. Since this information is fundamental for various analyses and optimizations for object-oriented languages, there is a large body of work on class analysis; summaries of most of this work are available in [11, 22].

A *call chain* is a sequence of call graph edges $e_1, \ldots, e_k$ such that the target of e_i is the same as the source of e_{i+1} for $1 \leq i < k$. Such a chain may be thought of as an abstraction of the top of the run-time call stack. For example, chain $e_1 = (m, n); e_2 = (n, p)$ represents a call stack configuration in which the top element on the stack is method p, with n and m under it. A call chain is *feasible* if there exists a run-time execution during which the corresponding configuration of the call stack occurs at least once.

Static analysis of call chains computes a set of chains that is a conservative estimate of the actual chains that may be observed at run time. *Dynamic analysis* of call chains constructs the set of chains observed during a particular execution. As discussed in Section 2, both categories of analyses provide valuable information for program understanding and software testing.

1.1 Static and Dynamic Analysis of Call Chains

In this paper we consider static call chain analyses that take as input a call graph computed by some class analysis algorithm. The designers of a call chain analysis have to address several important issues. First, how to ensure that the analysis reports few (if any) infeasible chains? In the simplest case, the analysis could just construct all edge sequences in the given call graph. However, some sequences may be infeasible due to infeasible call graph edges. Even if all call edges are feasible, certain sequences may still be infeasible; an example of this situation is presented in Section 3. Existing work does not provide direct evaluations of the degree of call chain imprecision and the sources of this imprecision. In software tools, such imprecision leads to wasted time and effort for tool users. For example, in software maintenance, an infeasible chain reported by the tool may require time-consuming manual investigation of the code. Similarly, as part of test coverage requirements, an infeasible chain "pollutes" the coverage metrics and wastes valuable tester time.

Analysis designers need to address various other issues:

- The results should represent the set of call chains in a compact manner. The representation should be linear in the number of chains, and should be easy to use in a subsequent dynamic analysis.
- The analysis should employ mechanisms for controlling the number of reported chains (e.g., by filtering out chains that are too long), and should be able to handle the infinite number of chains in the presence of recursion.
- Different levels of call edge granularity should be supported—for example, an edge may either represent a relationship "m_i invokes m_j", or a finer-grain relationship "call site c_k inside m_i invokes m_j".

ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.

For the purposes of this paper, a dynamic call chain analysis takes as input a set of call chains computed by some static call chain analysis, and reports the run-time coverage of these chains. The analysis uses code instrumentation that triggers certain run-time events. A dynamic call chain analysis should have several desirable properties:

- It should instrument *only* the component of interest, and not other components that interact with it. This reduces the intrusiveness of the approach and the run-time instrumentation overhead.
- It should identify and ignore run-time events that are not relevant to the given static call chains.
- It should handle correctly the potential interactions between the instrumented and the non-instrumented parts of the program. For example, the analysis should take into account the possibility of callbacks to the instrumented component, and should keep track of the level of reentrance with respect to these callbacks.

1.2 Analysis Framework

To address these issues, we have defined a framework for performing static and dynamic analysis of call chains in Java software. The approach is based on a data structure that is a generalized form of a *calling context tree* [2]. This tree encodes a set of static call chains, and allows subsequent dynamic analyses to track the execution of these chains. The static analysis for tree construction is parameterized by the (1) call graph construction algorithm, (2) granularity of call graph edges, (3) mechanism for choosing "interesting" chains, and (4) handling of recursion. This parameterization allows various uses of the approach in software tools.

We have performed a set of experiments that evaluate the imprecision for several instances of the framework. To the best of our knowledge, these are the first available empirical results that directly measure call chain imprecision. This empirical study addresses two key questions. First, what *exactly* is the imprecision in the chains computed using popular class analyses such as Rapid Type Analysis [4] and Andersen-style points-to analysis [3, 27, 15, 19, 14]? Second, what are the limits of call chain precision *for all class analyses*—that is, how much call chain infeasibility would remain even if the "most powerful" class analysis were available? The answers to these questions provide valuable insights for analysis designers and tool builders. Our results lead to some surprising conclusions and directions for future work.

1.3 Test Coverage Tool

Using the analysis framework, we have built a coverage tool for testing of Java software. The tool uses a set of *coverage criteria* that can be defined as instances of the framework. The criteria are related to several testing approaches proposed in previous work, as discussed in Section 5. Using the dynamic analysis, the tool keeps track of run-time chain coverage achieved during test execution, and reports coverage metrics for the different criteria.

We present a study of applying the tool to the Mauve test suite (`sources.redhat.com/mauve`). Mauve is an open-source suite of tests for the standard Java libraries. We used the coverage tool to evaluate and improve Mauve. In our experience, the tool can expose serious weaknesses in the tests, and can provide valuable guidance for creating more comprehensive test cases.

1.4 Contributions

The contributions of this work are:

- a parameterized framework for static and dynamic analysis of call chains in Java components
- a call-chain-based test coverage tool, and a study of applying the tool to a real-world test suite
- a precise evaluation of the imprecision of static call chains in the context of the study, and insights about the inherent precision limits of all class analyses

1.5 Outline

The rest of the paper is organized as follows. Section 2 discusses the use of call chains for software understanding and testing. Section 3 defines the static analysis framework. The dynamic analysis is described in Section 4. Section 5 presents the design of the coverage tool, and Section 6 defines our approach for precision evaluation. The empirical study is described in Section 7. Section 8 discusses related work, and Section 9 presents conclusions and directions for future work.

2. USES OF CALL CHAIN INFORMATION

2.1 Software Understanding

Call graphs provide a natural representation of the calling relationships among methods. Tools for code browsing and understanding can provide graphical interfaces for displaying and navigating such graphs. This functionality can be strengthened by static call chain analysis that identifies and labels some of the infeasible call chains. For example, a call chain analysis can provide more precise answers for questions such as "given some component, how can a particular internal private method be reached from the component's interface?". Furthermore, information about static call chains is also necessary as part of other analyses. For example, reverse engineering of UML interaction diagrams [21] requires precise identification of feasible call chains. Similarly, the exploration of transitive dependencies in concern graphs [17] can benefit from more precise call chain information. Certain forms of interprocedural def-use analysis [26] also depend on good-quality call chain information. Dynamic analysis of call chains can also be used in software tools to enhance program comprehension. By determining the set of hot chains (i.e., chains that are most frequently executed), the analysis can provide profiling information that allows a programmer to understand better the behavior of her code.

2.2 Software Testing

Call chains can serve as basis for integration testing of both procedural software [16] and object-oriented software [5]. Object-oriented style of design and programming results in many small methods, with a large number of messages exchanged between objects. Various authors have argued that in this context it is important to perform comprehensive testing of calling relationships. Jorgenson and Erickson [13] propose object integration testing that requires coverage of all relevant call chains (referred to as method-message paths). Binder [5] discusses integration testing (e.g., top-down integration and collaboration integration) that requires lower-level modules to be exercised by tests written for a

top-level module. In this case, the necessary coverage can be achieved by choosing and exercising call chains that extend from the top-level modules to the lower-level modules. A similar approach is proposed for testing of layered architectures, where a test driver exercises the interface of the top layer [5]. Several authors [5, 1, 7, 8, 30] have proposed testing approaches based on UML interaction diagrams [21]. For example, coverage may be required for all possible beginning-to-end message sequences in a diagram. To achieve such coverage, a necessary condition is to cover all corresponding call chains.

3. STATIC ANALYSIS

3.1 Analysis Input

To allow the use of the analysis on incomplete programs (e.g., class libraries), we assume that it takes as input a set of Java classes that form the analyzed component. We also assume that some form of class analysis had been already used to compute a *component-level call graph*. Each non-abstract method in the component is represented by a graph node. Graph edges represent possible calling relationships. If a whole program is not available, some form of fragment class analysis [20] can be used to compute the graph. An example of a component-level call graph is shown in Figure 2.

In general, call graphs may represent different kinds of relationships:

- method m_i invokes method m_j
- call site c_k inside m_i invokes m_j
- call site c_k inside m_i invokes m_j on an instance of X

These different granularities are useful for different applications. For example, the display of call graphs in a software understanding tool may use the first category of relationships in order to simplify the displayed diagram. On the other hand, testing that takes into account polymorphism (e.g., [5]) requires information about individual call sites and the receiver classes at these sites. Our static and dynamic analyses are designed for the last choice from the list; therefore, we assume that each call graph edge is labeled with a pair (c_k, X) of a call site and a receiver class. Of course, if static and dynamic call chains are computed at this finest level of granularity, the results can be trivially projected to the other two approaches. The experiments in Section 7 present results for all three levels of granularity.

The static analysis and the subsequent dynamic analysis need to take into account the calling relationships between the component and code that is outside of the component. For convenience, we assume that this information is encoded in the given call graph through artificial nodes and edges. Artificial node *caller* is an abstraction of all external callers of the component, and artificial node *callee* is an abstraction of any external code that is called by the component. All edges from *caller* are artificial edges that represent calls to the component. Similarly, the edges to *callee* represent calls from the component.

This representation is conceptual and is used only to simplify the discussion; there is no need to explicitly construct these nodes and edges. Figure 2 shows an example of such a representation; the two artificial nodes are shown with shaded rectangles. For every component method m that may be called from the outside, there must be at least one artificial edge from *caller* to m. Similarly, if m contains a call site that may invoke some external method for some receiver class, the graph should contain an artificial edge from m to *callee* with the corresponding label.

```
public abstract class NumberFormat {
   public int getMaxIntDigits() { return maxIntDigits; }
   public void setMaxIntDigits(int n)
      { maxIntDigits = n; }
   private int maxIntDigits = 40;
}
final public class FormatSymbols {
   FormatSymbols() { separator = '?'; }
   public char getSeparator()
      { if (separator == '?')  return getDefault();    (c1)
        else return separator; }
   public void setSeparator(char c) { separator = c; }
   public char getDefault() { return ','; }
   private char separator;
}
public class DecimalFormat extends NumberFormat {
   public DecimalFormat()
      { symbols = new FormatSymbols(); }               (c2)
   public FormatSymbols getSymbols() { return symbols; }
   public String toPattern()
      { return toPattern(false); }                     (c3)
   public String toLocalizedPattern()
      { return toPattern(true); }                      (c4)
   private String toPattern(boolean localized) {
      StringBuffer res = new StringBuffer();           (c5)
      if (getMaxIntDigits() == 0)                      (c6)
                return res.toString();                 (c7)
      res.append(localized ?                           (c8)
                symbols.getSeparator():SEPARATOR);     (c9)
      return res.toString();                           (c10)
   }
   private FormatSymbols symbols;
   private static final char SEPARATOR = ',';
}
```

Figure 1: Sample classes based on package `java.text`.

3.2 Running Example

We illustrate the analysis using the sample component in Figure 1. The component contains classes based on code from the standard package `java.text`. `NumberFormat` is the root of a hierarchy for parsing and formatting of numbers. `DecimalFormat` implements this functionality for decimal numbers. `FormatSymbols` stores symbols that are necessary for locale-specific formatting. We considered some of the code from package `java.text`, and through simplifications and modifications obtained the sample code in Figure 1. For ease of explanation, labels c_i are attached to all call sites.

Figure 2 shows a possible call graph for this component. For this example, we assume that all public methods in the component may be called by non-component code, and therefore the graph contains edges from *caller* to these methods. These artificial edges have empty labels. The remaining edges in the graph are labeled with the corresponding call site; in this example, the receiver classes are obvious and are not shown explicitly as part of edge labels. Note that all call graph edges in Figure 2 are feasible—that is, it is possible to exercise each edge at run time.

3.3 Calling Context Trees

A *calling context tree* is a data structure originally proposed by Ammons et al. [2] for the purposes of context-sensitive profiling. The root of the tree represents the start-

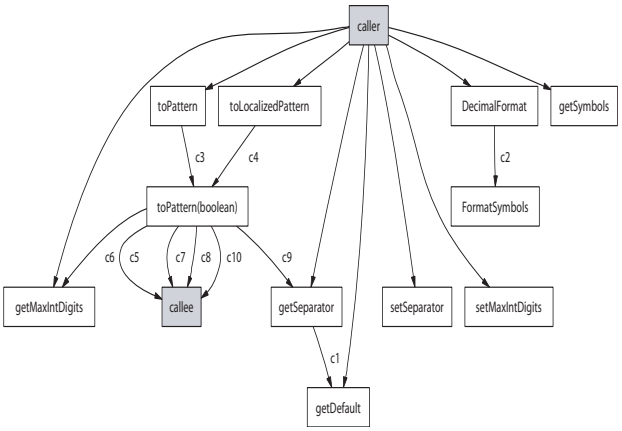

Figure 2: Component-level call graph.

```
ContextForest forest;
boolean use_backedges;
proc buildTree(Method entry)
1     TreeNode root = forest.createNewRoot(entry);
2     build(entry,root,new EdgeList());
proc build(Method m, TreeNode t, EdgeList chain)
3     for each e ∈ OutgoingCallGraphEdges(m)
4        if (stop(chain,e)) continue;
5        Method m′ = target(e);
6        if (not inComponent(m′)) continue;
7        if(use_backedges and m′ ∈ chain)
8           forest.createNewBackedge(t,m′,label(e));
9           continue;
10       TreeNode t′ = forest.createNewChild(t,m′,label(e));
11       chain.append(e);
12       build(m′,t′,chain);
13       chain.removeLastElement();
```

Figure 3: Algorithm for tree construction.

ing point of the program, and each tree node n represents a particular call chain containing n and all of its ancestors in the tree. Edges represent relationships of the form "call site c_k inside procedure p_i invokes procedure p_j". The data structure is built at run time, and is used to gather frequency information and other dynamic metrics for call chains.

We have modified the approach from [2] in several dimensions. First, since we are interested in analyzing a component instead of a complete program, we associate a separate calling context tree with each *entry method* in the component. In the component-level call graph, an entry method has an incoming artificial edge which shows that the method may be called by non-component code. Thus, we create a forest of context calling trees. Second, we build the trees using static analysis, which allows their use for various applications: for example, to define test coverage requirements, to represent possible calling behavior in program understanding tools, and to provide call chain information to other static analyses. As described in Section 4, a subsequent dynamic analysis can traverse the trees at run time in order to gather coverage information and other metrics. Third, since the set of all possible static call chains could be large, the analysis uses a "stopping" mechanism to restrict the set of represented call chains and to filter out some chains that are infeasible. Furthermore, we allow different treatment of recursion. Finally, since we are interested in analysis of object-oriented software, we augment tree edges with information about the receiver class of the invocation.

3.4 Tree Construction

Figure 3 shows the algorithm for constructing a context calling tree for an entry method. The algorithm maintains a list of call graph edges that correspond to the current chain, and attempts to extend the chain with a new edge. We use `label(e)` to denote the pair (c_k, X) of a call site and a receiver class associated with call graph edge `e`. The call to `createNewChild` at line 10 creates a new tree node corresponding to $\mathtt{m}'$, and adds it as a child of tree node t; the newly created tree edge is labeled with `label(e)`.

Boolean function `stop` used at line 4 encodes a *stopping criterion* for building the tree. Such a criterion is a parameter of the analysis and can play two roles. First, it allows filtering of call chains that are not "interesting" for the intended uses of the analysis. For example, if the chains are used to define test coverage requirements, this filter can control the strength of the requirements. Similarly, for reverse engineering of UML interaction diagrams (e.g., [29]), the stopping criterion can filter out all calls that are not part of the object interactions under consideration.

Another use of the filter is to prevent the creation of infeasible call chains. If it can be determined that a call graph edge `e` is infeasible under the calling context represented by `chain`, the edge can be ignored. In our implementation, we use the following simple infeasibility test: for any call through `this`, the receiver class associated with `e` should match the receiver class associated with the last edge in `chain`. Of course, more sophisticated approaches are possible—for example, context-sensitive class analyses such as k-CFA [23, 11] can provide information that allows filtering of some infeasible chains.

Another parameter of the analysis is the treatment of recursion, encoded by boolean flag `use_backedges`. The original definition by Ammons et al. represents recursion by creating a backedge from a tree node to some ancestor of that node.[1] If `use_backedges` is true, our analysis uses the same approach (lines 7–9). The call to `createNewBackedge` at line 8 traverses the ancestors of `t`, finds the ancestor node $\mathtt{t}'$ that corresponds to $\mathtt{m}'$, and creates a backedge from `t` to $\mathtt{t}'$ labeled with `label(e)`. If the flag is false, recursion is unrolled until the stopping criterion indicates that no further unrolling is necessary; of course, the criterion must guarantee that the unrolling eventually terminates. Such unrolling is useful for testing of recursion when the call chains represent test coverage requirements. For example, if method `m` invokes itself, then the inclusion of k consecutive occurrences of `m` in a call chain indicates that during testing, at least $k-1$ repetitions of the recursion must be executed.

Example. Figure 4 shows part of the forest of calling context trees for the example from Figure 2, with stopping criterion `chain.size()==2`. The creation of the dotted edges to `getDefault` is prevented by this stopping criterion. Calls to external code, represented in Figure 2 as artificial edges to *callee*, are not added to the trees due to the check at line 6. Note that call chain (n_1, n_2, n_4) is infeasible: regardless of how the component is invoked, the chain cannot be exercised because `localized` is false.

[1] Thus, the data structure is not strictly a tree. Due to the clear separation between tree edges and backedges, this is not an issue.

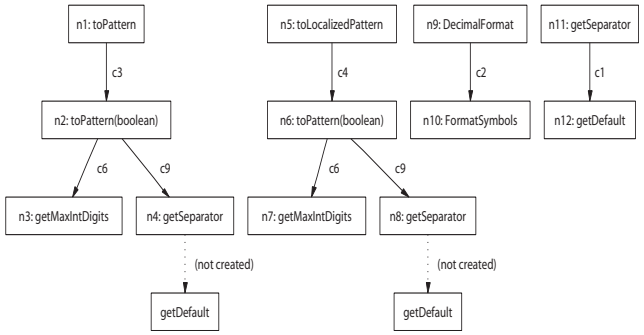

Figure 4: Examples of context calling trees.

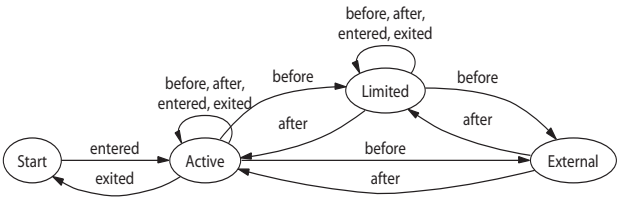

Figure 5: Analysis states and transitions.

4. DYNAMIC ANALYSIS

This section presents a dynamic call chain analysis which takes as input the calling context trees for all entry methods, performs run-time traversals of these trees, and gathers dynamic information related to the underlying call chains. We focus the discussion on the approach for traversing the trees. The run-time information gathered during these traversals depends on the intended use of the dynamic analysis. For example, frequency information may be collected for the purposes of program understanding and performance optimization. Section 5 describes the information gathered for test coverage tracking in our coverage tool.

4.1 Instrumentation

The dynamic analysis employs two kinds of code instrumentation. First, we need instrumentation at the beginning and at the returns of each entry method. This instrumentation creates run-time events of the form *entered*(m) and *exited*(m), where m is an entry method. Second, we need instrumentation immediately before and immediately after certain call sites. For a call site c, the generated run-time events are *before*(c, X) and *after*(c, X), where X is the class of the receiver object. At run time, X can be obtained through reflection. If c is a call to a static method, we use $X = none$. Call site instrumentation is needed only for call sites that correspond to edges in the input forest, and for call sites that have outgoing call edges to non-component code (i.e., *exit call sites*).

Example. For the code in Figure 1 and the trees in Figure 4, method entry/exit instrumentation is inserted for all public methods. Call site instrumentation is inserted for $\{c_1, c_2, c_3, c_4, c_6, c_9\}$ because they are included in the forest, and for $\{c_5, c_7, c_8, c_{10}\}$ because they are exit call sites (i.e., they are labels of artificial edges to *callee* in Figure 2).

The instrumentation plays two roles. Before/after events at call sites in the forest allow the dynamic analysis to move upwards and downwards in the trees. Events at entry methods and at exit call sites are used to keep track of how the run-time execution enters and leaves the component.

The dynamic analysis was specifically designed to use instrumentation *only inside the component*. The rest of the system is not affected, which makes the technique less intrusive and easier to use in practice. The alternative is to require instrumentation inside callers and/or callees of the component; however, this would make the analysis harder to use in large and evolving systems. For example, with this alternative approach, changes in callers/callees will require their re-instrumentation; furthermore, software configuration management will become more complicated because it has to track both the original and the instrumented versions of many components. Another design goal was to *avoid unnecessary instrumentation* in the component—for example, for call sites inside the component that are not related to the trees or to the component boundaries. This is important when we have a large component, but the call chains of interest only cover a small subset of this component.

4.2 Dynamic Analysis States

We describe the operation of the analysis as transitions among four internal analysis states. These states represent certain properties of the run-time execution that is being tracked by the analysis. State *Start* represents periods in the execution when no component methods are active on the run-time call stack. State *Active* corresponds to moments in which the currently active method is represented by a node in the forest. We use *Active.curr* to denote the tree node that corresponds to the currently active method. State *Limited* is used when the currently active method is a component method that is not represented in the forest. State *External* models the case when the current method is not in the component, but it was invoked directly or transitively by a component method.

Figure 5 summarizes the states and the transitions among them. The dynamic analysis algorithm is based on these transitions and on some additional information encoded in counters and stacks, as discussed shortly. Figure 5 and the remainder of this section describe the case in which component reentrance does not occur. Section 4.3 discusses the generalization of the approach in order to handle reentrance.

4.2.1 Start State

When the dynamic analysis is activated, the initial state is *Start*. At this moment, there should be no component methods on the run-time call stack. If the state is *Start* and a run-time event *entered*(m) is observed, the state is changed to *Active*, and the current node *Active.curr* is initialized with the root of the tree for entry method m.

4.2.2 Active State

While in state *Active*, the analysis uses the run-time events to decide how to traverse the tree. Event *before*(c, X) moves *Active.curr* down one level in the tree, by following the tree edge labeled with (c, X). The algorithm in Figure 3 guarantees that there is at most one edge with this label coming out of *Active.curr*. If no such edge exists, the state is changed to *Limited* or *External* as described below. Intuitively, run-time event *before*(c, X) means that the flow of control is about to enter the corresponding callee method under calling context (c, X). Event *after*(c, X) means that

the flow of control has just returned back to the caller; thus, the analysis follows the corresponding tree edge backwards, and *Active.curr* is moved one level up in the tree.

If `use_backedges` was turned on for the static analysis, event *before*(c, X) may correspond to a backedge. In this case, the backedge is followed and *Active.curr* is set to the appropriate ancestor node n. Furthermore, the backedge is pushed onto a stack associated with n. Subsequently, when *Active.curr* $= n$ and the analysis observes an *after* event corresponding to the top of n's stack, the backedge is popped from the stack and *Active.curr* is moved to the backedge source.

Event *exited*(m) triggers a transition *Active* $\rightarrow$ *Start* when the execution of entry method m completes, *Active.curr* is the root of the corresponding tree, and the stack of backedges for the root is empty.

While in the active state, events *entered* and *exited* may also be observed. This happens, for example, when one entry method calls another entry method. In Figure 5, this possibility is represented by *entered* and *exited* associated with the transition *Active* $\rightarrow$ *Active*. Such events should be ignored by the analysis, with the exception of *exited* at the root of the tree.

Example. In Figure 1, consider the invocation of method `getSeparator` by some code external to the component. Suppose that the dynamic analysis takes as input the trees in Figure 4 and the single-node trees for the remaining entry methods. From state *Start*, event *entered*(*getSeparator*) triggers a transition to *Active* with *Active.curr* $= n_{11}$. Event *before*$(c_1, \mathit{FormatSymbols})$ changes *Active.curr* to n_{12}. Since `getDefault` is an entry method, the corresponding events *entered* and *exited* are observed and ignored by the analysis. After `getDefault` completes, the execution returns to c_1 and *after*$(c_1, \mathit{FormatSymbols})$ occurs. This changes *Active.curr* back to n_{11}. The return from `getSeparator` generates event *exited*(*getSeparator*), which triggers *Active* $\rightarrow$ *Start*.

4.2.3 Limited State

Due to the stopping criterion in the static analysis, some of the run-time call chains may be missing from the forest. To handle this case, we introduce an analysis state *Limited* to represent execution states in which the current active method is in the component, but there is no corresponding tree node. For example, for the trees shown in Figure 4, an invocation of `toLocalizedPattern` may eventually lead to a call to `getDefault`. However, node n_8 does not have a child corresponding to this call. Therefore, when *Active.curr* $= n_8$, an event *before*$(c_1, \mathit{FormatSymbols})$ will trigger a transition *Active* $\rightarrow$ *Limited*. Note that a child may also be missing because the target method is outside of the component (line 6 in Figure 3). For example, for Figure 4, *Active.curr* $= n_6$ and event *before*$(c_{10}, \mathit{StringBuffer})$ mean that the flow of control is about to leave the component. In this case a transition *Active* $\rightarrow$ *External* is necessary, as described shortly. To distinguish between these two cases, the static analysis associates with the forest the set of call graph edge labels that correspond to calls to component methods. The dynamic analysis uses this set to choose the appropriate state transition.

If we encounter an event *before*(c, X) that triggers a transition *Active* $\rightarrow$ *Limited*, this "freezes" the traversal. To resume the traversal, the analysis needs to perform a transition *Limited* $\rightarrow$ *Active*. In general, we cannot resume the traversal at the first event *after*(c, X). The reason is that additional events *before*(c, X) may happen in state *Limited*, and the first *after*(c, X) will correspond to the last event *before*, not to the original one.

To solve this problem, we introduce a counter associated with state *Limited*. On transition *Active* $\rightarrow$ *Limited*, the counter is set to 1. For every event *before*(c, X) which invokes a component method, the counter is incremented. Every *after* event decrements the counter. When the counter becomes zero, a transition *Limited* $\rightarrow$ *Active* occurs. Note that if a *before* event corresponds to the invocation of a non-component method, we need to perform a transition *Limited* $\rightarrow$ *External* instead of staying in state *Limited*.

As with state *Active*, in state *Limited* it is possible to observe events *entered* and *exited* due to entry methods being invoked by other component methods. In Figure 5, this situation is represented by *entered* and *exited* associated with the transition *Limited* $\rightarrow$ *Limited*. All such events should be ignored by the analysis.

Example. Consider node n_8 in Figure 4. Upon event *before*$(c_1, \mathit{FormatSymbols})$, the state becomes *Limited* and the execution enters `getDefault`. For the sake of this example, suppose that `getDefault` were mutually recursive with `getSeparator`. If the run-time recursion went through k iterations, there would be k events *before*$(c_1, \mathit{FormatSymbols})$, followed by $k+1$ events *after*$(c_1, \mathit{FormatSymbols})$. Only the last event should trigger the transition back to *Active*. While in *Limited*, events *entered* and *exited* for `getDefault` and `getSeparator` will be observed and ignored by the analysis.

4.2.4 External State

Analysis state *External* represents a state of the run-time execution in which the current active method is not a component method, but the call stack contains at least one component method. As described above, transitions *Active* $\rightarrow$ *External* and *Limited* $\rightarrow$ *External* will occur when the component code invokes a non-component method. Since we insert instrumentation only inside the component, no run-time events will be generated by the execution of code that is external to the component. If this code does *not* call component methods, then the only possible transition is back to *Active* or *Limited* upon receiving an *after* event. To know which of these two possible transitions to perform, the dynamic analysis has to remember the last state before entering *External*, and has to resume the traversal in that state.

Example. For the chains starting at n_1 and n_5 in Figure 4, event *before*$(c_{10}, \mathit{StringBuffer})$, puts the analysis in state *External*. After the execution of `toString` completes, *after*$(c_{10}, \mathit{StringBuffer})$ changes the state back to *Active*. Similar changes occur for call sites c_5, c_7, and c_8.

4.3 Component Reentrance

The approach described above cannot be applied in the presence of *component reentrance*—that is, run-time executions in which the component calls external code, and during the execution of this external code there is a call back to the component. The problem is that once a component is reentered, the information related to previous entrances is lost. For example, in Figure 1, suppose that `toLocalizedPattern` contained a call to some external method that in turn calls `toPattern`. During the execution of `toLocalizedPattern`, the dynamic analysis could enter state *External* and then observe event *entered*(*toPattern*). If this happens, the anal-

ysis should make a transition *External* → *Active* and should start traversing the tree rooted at `toPattern`. However, after `toPattern` returns, the state would become *Start* and all information related to the execution of `toLocalizedPattern` would be lost.

To solve this problem, the dynamic analysis maintains a *stack of traversals*. Each traversal corresponds to an entry into the component from some external code. When the component is entered, a new traversal is created and pushed onto the stack; upon return, the traversal is popped. To implement this approach, we define the notion of a *traversal* as a run-time object that has the following information:

- **state**: one of *Active*, *Limited*, *External*
- **curr**: current tree node
- **stacks**: stacks of backedges for all backedge targets
- **counter**: counter for state *Limited*
- **prev**: previous state, one of *Active*, *Limited*
- **level**: entry level, an integer ≥ 0

The counter is used to decide when to change the traversal state from *Limited* to *Active*, as described earlier. State **prev** is the traversal state before the traversal went into *External*. As discussed above, **prev** is needed in order to determine how to change traversal state when a call to external code returns. Integer **level** is the height of the traversal stack when the traversal was created. This *entry level* characterizes the depth of reentrance for the traversal.

The dynamic analysis starts at state *Start* with an empty traversal stack. Event *entered*(m) creates a new traversal and pushes it onto the stack. The traversal is in active state, with current node being the root of the tree for m, and with entry level 0. If the flow of control leaves the component, the traversal state becomes *External*. If subsequently event *entered*(n) is observed, a new traversal with entry level 1 is created and pushed onto the stack. Eventually, *exited*(n) is observed and the traversal is popped from the stack. After possibly many traversals with entry levels ≥ 1, the flow of control returns back to the first traversal. Event *exited*(m) pops the traversal from the stack (which leaves the stack empty), and sets the analysis state to *Start*. Any subsequent call to an entry method also triggers a sequence of traversal pushes and pops; upon completion of the sequence, the stack is empty and the analysis state is *Start*.

4.4 Restrictions

The dynamic analysis described above is designed under two restrictions. First, the run-time execution is assumed to be single-threaded. It may be possible to generalize the technique for multi-threaded programs by keeping multiple traversal stacks, one per run-time thread; we are currently working on this generalization. A second restriction is that the run-time execution should not involve exceptions that are caught by the component or by its callers. In order to handle such exceptions, the static analysis should include some form of exception flow analysis (e.g., similar to [24]). We plan to address this issue in the near future.

For the experiments presented in Section 7, we ensured that these restrictions were satisfied through a combination of static and dynamic techniques. Using static call graph analysis, we showed that the analyzed components were not reachable from the starting points of any threads. Since finalizers and static initializers (represented in bytecode by methods `<clinit>`) can be executed in multi-threaded fashion by the virtual machine, we also ensured that component methods were not reachable from non-component finalizers and `<clinit>` methods. For finalizers defined inside the component, instrumentation was inserted to warn about their execution. We generalized the dynamic analysis to handle the execution of static initializers defined in the component; this technique will be described in a companion technical report. All catch clauses in the component were instrumented to warn about exceptions being caught. Exceptions that would have been normally propagated to the callers of the component were caught by try-catch blocks inserted in the bodies of all entry methods. This combination of techniques ensured that the restrictions were satisfied for the experiments described in Section 7.

5. COVERAGE TOOL

Using the framework for static and dynamic analysis, we built a test coverage tool for Java. The tool takes as input a component under test (CUT) and a test suite T for this component. The static analysis is used to define test coverage requirements for the suite. During test execution, the dynamic analysis tracks the achieved run-time coverage. The calling context trees are a natural representation of coverage requirements for call chains: there is one tree node per chain (i.e., the representation is compact), and it it is easy to find the longest covered prefix of an uncovered chain, which is useful when constructing additional test cases.

The tool starts by building a component-level call graph using fragment class analysis [20], a class analysis technique designed to analyze incomplete programs. This is achieved by creating an artificial `main` method which calls all methods that are invoked by the given test suite and "simulates" all possible effects on object references. A whole-program class analysis is then executed on `main`, on the CUT, and on any code reachable from `main` or the CUT. The resulting call graph is used as input to the static analysis of call chains. This approach allows us to compute an over-estimate of all calling relationships that may be observed during the execution of *any test suite* T' such that T' invokes the same methods as the given suite T, and the execution of T' involves only classes that are available for analysis by the fragment class analysis. After running the class analysis, it is trivial to construct a call graph similar to the one in Figure 2.

Given the component-level call graph, the coverage requirements can be defined using the static analysis framework. In particular, the stopping criterion can be used to define the set of "interesting" call chains that should be covered. This could happen interactively—the tester can choose a set of relevant entry methods, the tool can display all possible chains of length 1 starting at these methods, the tester can filter out chains that are not relevant for his testing goal, and this can continue until the desired breadth and depth of the trees is achieved. Such a technique may be applicable, for example, in the context of collaboration integration [5]. This approach for integration testing is based on defining and testing a sequence of collaborations, where each collaboration can be defined (interactively) by a set of call chains. Similarly, interactive chain definition may be applicable for integration testing of layered systems [5].

An alternative approach is to compute all chains up to a certain length, and then (if necessary) to filter out irrelevant chains. This may be applicable, for example, when

performing testing based on UML interaction diagrams. In this case, a necessary condition for achieving high coverage of object interactions is to have high coverage of the corresponding call chains. Since interaction diagrams typically have limited depth of the represented calling relationships, the length of the call chains is also limited, and therefore the stopping criterion can be based on chain length.

In addition to defining the call chains that need to be covered, a coverage criterion may also define the *entry levels* at which they should be covered. This may be used for testing the behavior of the component under reentrance—that is, situations in which the component calls external code, and during the execution of this external code there is a call back to the component. As Szyperski points out [28], component reentrance creates the possibility of observing invalid intermediate states, in which case maintaining correctness becomes difficult. To ensure correct behavior in this case, coverage criteria may require a certain degree of reentrance to be exercised during testing. Thus, for each node in a calling context tree, a set of required *entry levels* may be defined. For example, if we require a chain $n_r, \ldots, n_s$ (n_r is an entry method) to be covered at entry level 1, the execution of the test suite must create the following configuration of the run-time call stack:

$$m_1, \ldots, m_{p-1}, n_p, \ldots, n_{q-1}, m_q, \ldots, m_{r-1}, n_r, \ldots, n_{s-1}, n_s$$

Here the bottom of the stack is shown on the left; methods m_i are external to the CUT, and methods n_j are in the CUT. A variety of coverage requirements can be defined in this manner: e.g., "n must be covered for some level $l > 0$", or "n must be covered for $l = 1$". Of course, such requirements may be infeasible for some tree nodes.

Recall from Section 4.3 that the dynamic analysis tracks precisely the depth of reentrance for each tree traversal. Therefore, for each tree node it is possible to know exactly the entry levels at which it was traversed at run time. This allows the coverage tool to support coverage requirements that take into account component reentrance, in order to ensure correct behavior of complicated interactions (e.g., callbacks) between the CUT and the rest of the system.

For the experiments presented later, we defined the coverage requirements in the following manner. Starting from the artificial `main` method, we considered all call chains containing at most k call graph edges. In each chain, every maximal subsequence of CUT-only methods—that is, subsequence with no adjacent CUT methods—was mapped to a corresponding chain in the calling context trees. (This required some trivial additions to the algorithm from Figure 3.) Thus, the stopping criterion was based on the chain length, counting from `main`. This approach was used because we plan to generalize the tool to support testing based on interaction diagrams, and k-limiting of chain length is the appropriate stopping criterion in this context. If within a single chain there were *multiple* CUT-only maximal subsequences, they represented potential component reentrance; the first subsequence in the chain defined coverage requirements at entry level 0, the second one at entry level 1, etc.

The stopping criterion also used the simple infeasibility test described in Section 3.4: for any call through `this`, the receiver class of the call graph edge matched the receiver class of the last edge in the chain. Since we were interested in achieving coverage of recursion during testing, the static analysis was run with recursion unrolling turned on (i.e., backedges were not used). Finally, we computed and measured coverage requirements at three different granularities: (1) method m_i invokes method m_j, (2) call site c_k inside m_i invokes m_j, and (3) call site c_k inside m_i invokes m_j on an instance of X. Choices (2) and (3) result in more comprehensive coverage requirements, while (1) produces a coarser-grain representation for which coverage may be easier to achieve.

6. STATIC ANALYSIS IMPRECISION

As discussed earlier, the static analysis described in Section 3 can produce infeasible call chains. For example, in Figure 4, (n_1, n_2, n_4) is infeasible even though all underlying call graph edges are feasible. Such infeasibility presents a serious problem for program understanding and testing, because it provides misleading information to tool users. In our coverage tool we had the opportunity to evaluate the degree of this imprecision. Through a somewhat labor-intensive process (about one person-month), we wrote test suites that achieved *the highest possible coverage* for several coverage criteria. Substantial effort was made to ensure that the best possible coverage was achieved. For each component, one of the authors constructed (1) a suite that he believed to cover all feasible chains, and (2) arguments of why each of the remaining chains could not be covered. The other author then considered all chains not covered by the suite, and independently constructed his own arguments of infeasibility. The two authors then examined each other's arguments. After coming to an agreement for each of the uncovered chains, these chains were deemed infeasible and due to the imprecision of the static analysis.

The experiment described above allows *exact* evaluation of the precision of the static call chain analysis. Since precision is a critical issue for the use of any static analysis in software engineering tools, such results provide valuable insights for analysis designers and tool users. Traditionally, static analysis research evaluates analysis precision by comparing several static analyses with each other. Some researchers have also compared static analysis results with information collected at run time for some set of program inputs. However, neither of these approaches provides a precise answer to the the key question: what is the difference between the static analysis solution and the "perfect" solution? Our results provide a *precise answer* to this question.

Clearly, this evaluation approach has certain limitations. For example, a potential validity threat is the possibility of human error when determining the perfect solution. Another obvious concern is the scalability of such evaluations. Nevertheless, we believe that this approach is valuable because it leads to insights that cannot be obtained through traditional techniques. More in-depth discussion of this topic is available in [18].

For analysis designers, understanding why analyses are imprecise is crucial for future improvements. Our goal was to evaluate the limits of current class analysis technology with respect to the computation of call chains. We considered one particular source of imprecision: the standard approach of imprecise modeling for conditions in `if`, `switch`, `while`, etc. For example, at `if(p*q>5)..`, class analysis does not attempt to track the values of `p` and `q` and to decide whether the condition is true or false; rather, the actual condition is abstracted away and both the true and the false outcome are considered possible. Some class analyses may

attempt to track conditions involving reference values: e.g., `(x instanceof Y)`, `(x == y)`, or `(x == null)`. However, class analyses typically do not track conditions involving non-reference values.

We extended our study to estimate the degree of imprecision due to this common approximation. To achieve this, we defined an *abstracted semantics* for Java which differs from the standard semantics only in its treatment of certain conditions. An *irrelevant condition* is a condition which does not involve the reference equality operators `==` and `!=` [10, Section 15.21.3] and the type comparison operator `instanceof` [10, Section 15.20.2]. In the abstracted semantics, each run-time evaluation of an irrelevant condition results in a random choice between true and false. Compared with the standard semantics, the only difference is the choice of the next statement to be executed: after evaluating an irrelevant condition, rather than using the resulting boolean value, the execution randomly chooses one of the two possible next statements. The semantics of `switch` statements is modified in a similar fashion.

The abstract semantics captures the common approximation discussed earlier: all conditions that do not involve reference values (i.e., `instanceof`, or `==` and `!=` for references) are abstracted away. In this model, there may be multiple possible valid executions for the same program on the same input. If a call chain occurs during one of these valid executions, this means that *any* static analysis which abstracts away conditions involving non-reference values *must* report the chain as feasible (due to the safety of the analysis). This theoretical model allows us to draw conclusions about all possible static analyses that employ this approximation. Thus, if a call chain is infeasible in the standard Java semantics, but is feasible in the abstracted semantics, it will be erroneously reported as feasible by any such analysis. We refer to such chains as *worst-case chains*.

Example. Consider chain (n_1, n_2, n_4) in Figure 4. This chain is feasible in the abstracted semantics because both outcomes of `(localized ? : )` are possible regardless of the value of `localized`. If a static analysis uses the standard approximation described above, it will not be able to detect this chain as infeasible.

In our experiments we considered all chains that were reported by the static analysis from Section 3, based on a call graph computed by an Andersen-style fragment class analysis. For each chain that was actually infeasible (i.e., it could not be covered by our test suites), we attempted to construct manually an execution in the abstracted semantics for which the chain was feasible. Any chain for which such an execution was constructed successfully is a worst-case chain.

7. EMPIRICAL STUDY

For our experiments we used the components summarized in Table 1. These components were based on some of the test cases from the Mauve open-source test suite (`sources.redhat.com/mauve`). We considered six sets of related test cases and the functionalities they exercised. For each functionality, we defined a corresponding component containing the classes that implemented this functionality.

The components include classes from standard packages `java.util.zip` and `java.text`. The first two components are related to processing of GZIP and ZIP files, respectively. The remaining components contain functionality for text collation, formatting of dates, formatting of decimal numbers, and identifying boundaries in text. Column (2a) in Table 1 shows the number of classes in the CUT, and (3a) shows the number of methods in these classes. Column (2b) contains the number of classes that are directly or transitively referenced by CUT classes. The number of methods in classes from (2b) is shown in (3b).

(1) Component	(2)#Classes		(3) #Methods	
	(a) CUT	(b) All	(a) CUT	(b) All
`gzip`	6	199	41	2316
`zip`	5	195	40	2277
`collator`	12	203	160	2394
`date`	7	205	143	2504
`decimal`	7	198	141	2360
`boundary`	8	199	83	2302

Table 1: Analyzed Components.

Our implementation is built on top of the Soot framework (`www.sable.mcgill.ca`), and employs RTA [4] and the Andersen-style points-to analysis from [19] to compute the component-level call graph. Using different values of parameter k and different granularities of the call chains, we computed the coverage criteria as described in Section 5. For each node in the forest, the coverage requirements define the set of entry levels at which the node should be covered. We considered each pair of a node and an entry level for the node to be a separate chain. The number of such chains was measured in several ways. First, we measured the number of chains covered by the Mauve tests. Next, these tests were enhanced to achieve the highest possible coverage, as discussed in Section 6; this process revealed all feasible chains in the coverage requirements. The number of feasible chains was compared with the number of chains reported by two versions of the static analysis, one using Andersen's analysis and the other using RTA.

The results from these experiments are shown in Table 2. For each component, there are two rows labeled (a) and (b). Row (a) corresponds to trees in which edges represent relationships "call site c_k inside m_i invokes m_j on an instance of X", while row (b) is for trees with edges representing relationships "m_i invokes m_j". A third possibility is to distinguish call sites without taking into account receiver classes. The results for this variation are not included in the table because they were the same as the results from rows (a).

Part (1) of Table 2 shows the number of feasible call chains for $k = 1, 2, 3, 4$. Part (2) lists the number of feasible chains that were *not* covered by the Mauve tests. For example, for `collator(a)` and $k = 4$, 118 out of the 181 feasible chains were not covered. Parts (3) and (4) show the number of *infeasible* call chains that were reported by the static analysis, using RTA and Andersen's analysis to compute the call graph. For example, for `collator(a)` and $k = 4$, RTA reported 238 possible chains of which 57 were infeasible. Since RTA is less precise that Andersen's analysis, the numbers in part (3) are greater than or equal to the corresponding numbers in part (4).

7.1 Evaluation of Mauve

For several components, the coverage achieved by Mauve tests indicates possible test deficiencies. We investigated the chains that were not covered, as part of the effort to write additional test cases that achieved the highest possible coverage. For example, for `decimal`, some of the uncovered

		(1) Feasible				(2) Mauve				(3) RTA				(4) Andersen				(5) InfPrefix	
		1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	3	4
`gzip`	(a)	3	9	20	47		-1	-3	-16		+1	+4	+13			+1	+3	0%	100%
`gzip`	(b)	3	9	20	36		-1	-3	-9		+1	+4	+13			+1	+3	0%	100%
`zip`	(a)	2	7	25	43		-2	-6	-14			+1	+9			+1	+8	0%	43%
`zip`	(b)	2	7	15	21		-2	-6	-11			+1	+7			+1	+6	0%	60%
`collator`	(a)	8	36	105	181		-14	-63	-118		+3	+40	+57		+3	+31	+37	36%	33%
`collator`	(b)	8	22	41	66		-5	-11	-19			+10	+18			+1	+2	0%	0%
`date`	(a)	14	20	26	33								+7						
`date`	(b)	14	20	26	33								+7						
`decimal`	(a)	21	57	159	208	-2	-15	-45	-61	+1	+3	+72	+153		+2	+71	+152	0%	1%
`decimal`	(b)	21	48	127	166	-2	-7	-27	-41	+1	+2	+32	+54		+1	+31	+53	0%	5%
`boundary`	(a)	10	30	107	194	-1	-4	-12	-36	+1	+3	+3	+3						
`boundary`	(b)	10	28	63	85	-1	-3	-6	-8	+1	+2	+2	+2						

Table 2: Number of call chains.

chains were due to insufficient testing of certain boundary values such as positive infinity, Not-a-Number (NaN) [10], and double floating-point numbers with at least 127 digits before the decimal point. As another example, some of the chains in `collator` were not covered due to the lack of testing of the full set of decomposition rules for text comparison. The additional tests we wrote covered these chains and allowed us to distinguish between feasible and infeasible chains. Eventually, we plan to contribute our additional tests to the Mauve open-source repository.

7.2 Static Analysis Imprecision

Parts (1), (3), and (4) of Table 2 evaluate the imprecision of the static call chain analysis. For rows (a) and $k = 4$, 242 chains (25.5%) out of the 948 chains reported by the RTA-based analysis were infeasible; for the Andersen-based analysis, 200 chains (22.1%) out of the reported 906 chains were infeasible. For rows (b) and $k = 4$, 101 (19.9%) out of the reported 508 chains were infeasible for the RTA-based analysis, and 64 (13.6%) out of the reported 471 chains were infeasible for the Andersen-based analysis, The infeasible chains occur mostly in two components with complicated internal logic (`collator` and `decimal`).

Part (5) shows how many infeasible chains have infeasible prefixes, which is an indication of how many new chains required non-trivial effort to determine their infeasibility. For $k = 3$ and $k = 4$, we considered the number of infeasible chains introduced when going from $k-1$ to k in part (4). For example, for `collator(a)`, there are 28 new infeasible chains when k is increased from 2 to 3, and 6 new infeasible chains when k is increased from 3 to 4. If such a chain has a parent node (at the same entry level) that is also infeasible, the chain has an infeasible prefix. The corresponding columns in (5) show what percentage of the new infeasible chains have infeasible prefixes. For example, in `gzip`, all new chains introduced when k is increased from 3 to 4 have infeasible prefixes, while for `decimal` almost all new infeasible chains have feasible prefixes.

7.3 Worst-Case Call Chains

As described in Section 6, we used an abstracted semantics to identify infeasible chains that were worst-case chains. Surprisingly, for `collator(a)` and `decimal(a)`, all 189 infeasible chains reported by the Andersen-based analysis for $k = 4$ were determined to be worst-case chains. Thus, more powerful class analyses *would not have resulted in any precision improvements.* For these two components, using Andersen's analysis to compute the call graph provided the best possible call chain precision that may be expected from traditional class analysis approaches. Overall for the Andersen-based analysis with $k = 4$, 94.5% of the infeasible chains for rows (a) and 85.9% for rows (b) were worst-case chains.

This precision evaluation leads to an interesting result: for our subject components, Andersen's analysis is very close to the precision limits that may be expected from class analyses. We are not aware of any previous work that attempts to investigate such precision limits. Clearly, more experiments will be necessary to obtain similar measurements for additional subject components. If a large portion of the infeasible call chains in these experiments are worst-case call chains, this will imply that analysis designers have to look beyond traditional techniques for class analysis. For example, it may be necessary to investigate approaches that take into account context-dependent conditions that guard the execution of method invocations.

8. RELATED WORK

The closest related work to our investigation is by Souter and Pollock [25]. This work considers a particular source of call chain infeasibility, referred to as type infeasibility. This infeasibility is due to different types being propagated to the same polymorphic call site along different paths in the call graph. The work in [25] defines the conditions under which this infeasibility occurs, and presents an empirical evaluation of the number of type infeasible chains. The experiments provide indirect estimates of the number of such chains, and the precision of these estimates remains unclear. In contrast, our work provides a direct evaluation of the number of infeasible chains, and characterizes a source of imprecision which is independent of type propagation.

Context-sensitive class analyses employ some approximation of calling context, and can identify the infeasibility of a call graph edge under certain contexts. Context is modeled either through some abstract of the values of method parameters, or by the top k call sites on the call stack [11]. The scalability of most of these approaches remains unclear. Furthermore, as indicated by our investigation of precision limits, there may be other significant sources of imprecision that cannot be tackled with context sensitivity.

Some previous work proposes techniques for identifying infeasible paths in control-flow graphs, and for using this information to improve the precision of data-flow analysis [12, 9, 6]. Such approaches may be able to address some of the problems due to imprecise modeling of conditions. At

present, it is unclear how these techniques can be combined with class analysis to filter out infeasible chains.

9. CONCLUSIONS AND FUTURE WORK

We present a parameterized framework for static and dynamic analysis of call chains that can be easily adapted for various uses in software tools. Using the framework, we have built a coverage tool that supports several coverage criteria. The tool can be used for integration testing, for testing based on interaction diagrams, and for testing of component reentrance. Our experiments provide new insights about the degree of call chain analysis imprecision and the sources of this imprecision. The experiments confirm the value of investigations of analysis imprecision, and raise several interesting questions for future work.

Additional data, gathered by us and by other researchers, will be necessary to confirm the results of this study and to gain better understanding of the precision of call chain analysis. Even though obtaining such data is labor-intensive, it reveals important properties of different static analyses that affect the usefulness of these analyses in various software tools. Inclusion-based flow- and context-insensitive analyses appear to be a good starting point for such experiments: in our study, the Andersen-style analysis was capable of achieving precision very close to the theoretical limits.

Our results indicate that future investigations would benefit from evaluations of the theoretical limits of analysis precision, similar to the approach from Section 6. If more results confirm that substantial imprecision is introduced by abstracted modeling of conditions, traditional class analysis techniques will have to be extended to track (some) non-reference values. Several techniques provide possibilities for such extensions: for example, interprocedural constant propagation, or approaches for identifying infeasible paths in control-flow graphs. For imprecision that cannot be avoided even with such techniques, static analyses may have to be augmented to work with user assertions.

Acknowledgments. We would like to thank the ISSTA reviewers for their helpful comments.

10. REFERENCES

[1] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *Int. Conf. Unified Modeling Language (UML'00)*, pages 383–395, 2000.

[2] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Conf. Programming Language Design and Implementation*, pages 85–96, 1997.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[5] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[6] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Symp. Foundations of Software Engineering*, pages 361–377, 1997.

[7] L. Briand and Y. Labiche. A UML-based approach to system testing. *J. Software and Systems Modeling*, 1(1), 2002.

[8] F. Fraikin and T. Leonhardt. SeDiTeC—testing based on sequence diagrams. In *Int. Conf. Automated Software Engineering*, pages 261–266, 2002.

[9] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *Int. Symp. Software Testing and Analysis*, pages 80–94, 1994.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2 edition, 2000.

[11] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[12] L. H. Holley and B. Rosen. Qualified data flow problems. *IEEE Trans. Software Engineering*, 7(1):60–78, Jan. 1981.

[13] P. Jorgenson and C. Erickson. Object-oriented integration testing. *Commun. ACM*, 37(9):30–38, Sept. 1994.

[14] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Int. Conf. Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[15] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, June 2001.

[16] T. McCabe and C. Butler. Design complexity measurement and testing. *Commun. ACM*, 32(12):1415–1425, Dec. 1989.

[17] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Int. Conf. Software Engineering*, 2002.

[18] A. Rountev, S. Kagan, and M. Gibas. Evaluating the imprecision of static analysis. In *Workshop on Program Analysis for Software Tools and Engineering*, June 2004.

[19] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, Oct. 2001.

[20] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Trans. Software Engineering*, 2004.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[22] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Int. Conf. Compiler Construction*, 2003.

[23] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[24] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Software Engineering*, 26(9):849–871, Sept. 2000.

[25] A. Souter and L. Pollock. Characterization and automatic identification of type infeasible call chains. *Information and Software Technology*, 44:721–732, 2002.

[26] A. Souter and L. Pollock. The construction of contextual def-use associations for object-oriented systems. *IEEE Trans. Software Engineering*, 29(11):1005–1018, Nov. 2003.

[27] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an empirical comparison. Technical report, U. Passau, Sept. 2000.

[28] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[29] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from C++ code. In *Int. Conf. Software Maintenance*, pages 159–168, 2003.

[30] Y. Wu, M.-H. Chen, and J. Offutt. UML-based integration testing for component-based software. In *Int. Conf. COTS-Based Software Systems*, 2003.