# Control Flow Analysis for Reverse Engineering of Sequence Diagrams

Atanas Rountev Olga Volgin Miriam Reddoch
Department of Computer Science and Engineering
Ohio State University
{rountev,volgin,reddoch}@cis.ohio-state.edu

## ABSTRACT

Reverse engineering of UML sequence diagrams plays an important role in iterative development and in software maintenance. In static analysis for such reverse engineering, an open question is how to map control-flow graphs to the control-flow primitives of the recently-introduced new generation of UML. Our work presents the first algorithm that solves this problem. We also propose transformations of the reverse-engineered diagrams in order to improve diagram comprehension. Our extensive experiments show that these techniques are efficient and effective. We also describe a test coverage tool based on the sequence diagrams, and discuss its use for the evaluation of a real-world test suite.

## 1. INTRODUCTION

*Sequence diagrams* are essential UML artifacts for modeling the behavioral aspects of a system [22, 16]. The diagrams are particularly well-suited for object-oriented software, where they represent the flow of control during object interactions [9, 13]. A sequence diagram shows a set of interacting objects and the sequence of messages exchanged among them. The diagram may also contain additional information about the flow of control during the interaction, such as if-then conditions ("if `c` send message `m`") and iteration ("send message `m` multiple times") [22, 16]. An example of a sequence diagram is shown in Figure 2b.

### 1.1 Reverse-Engineered Sequence Diagrams

Modern iterative development allows quick adaptation by interleaving analysis, design, implementation, and testing within each iteration. The design is not defined completely in advance; rather, it evolves based on insights from the implementation and the testing done in the current and the previous iterations. In this context, it is often necessary to perform *reverse engineering* of the design from existing code. A typical scenario [13] is to perform design recovery in the beginning of the current iteration, by applying reverse engineering on the last iteration's code. The resulting design documents serve as the starting point for subsequent design work. Additional reverse engineering is also necessary during an iteration. As pointed out in one popular book on modern software development [13], in this context an important item on the "wish-list" for UML tools is reverse-engineering of sequence diagrams using current and correct UML notation. At present, there do not exist general static analysis techniques that achieve this goal.

Reverse engineering of sequence diagrams can also play an important role in the maintenance of large object-oriented systems. These systems are substantial investments that will have to be maintained for many years into the future, in the absence of the original designers and developers, and often with incomplete or non-existent design information. Reverse-engineered sequence diagrams provide essential insights for *software understanding and maintenance* of such systems, since object interactions are at the core of object-oriented design and programming.

Sequence diagrams are the basis for several approaches for *testing of object-oriented software* [3, 1, 4, 10, 28]. These approaches test the interactions among collaborating objects. Sequence diagrams, or the semantically-equivalent UML collaboration diagrams [22, 9], are used to determine the interactions that must be exercised. For example, it may be required to cover all relationships of the form "object `X` sends message `m` to object `Y`". Sequences of messages—for example, all possible beginning-to-end message sequences in the diagram—may also be considered for coverage. The use of such testing techniques can be simplified greatly by *coverage tools* based on reverse-engineered sequence diagrams. A coverage tool can use static analysis to extract one or more sequence diagrams from the tested code; the resulting diagrams represent the coverage requirements. The tool can then perform dynamic analysis during the execution of the given tests in order to obtain run-time coverage results.

### 1.2 Mapping to UML Control-Flow Primitives

First-generation UML (versions 1.x) provides limited features for representing the flow of control during an interaction [9, 3]. These deficiencies have been addressed by the designers of second-generation UML. The recently-introduced specification of UML 2.0 [16] defines a richer set of control-flow primitives to be used in sequence diagrams. For example, it becomes possible to represent in a more general manner alternative, repeating, and breaking behavior.

For reverse engineering of sequence diagrams, an important problem is how to map intra-method flow of control to the control-flow primitives of second-generation UML. Because the new version of UML was introduced only recently, no existing work addresses this problem. This means that reverse engineering tools cannot take advantage of the full expressive power of UML. This paper presents *the first static analysis algorithm* that performs this mapping. Given a control-flow graph for a method, our analysis builds a

data structure that represents this method's flow of control using UML 2.0 primitives. This data structure can be subsequently used as a building block in reverse-engineered sequence diagrams. Furthermore, the data structure provides a starting point for subsequent run-time coverage analysis in coverage tools that support testing based on reverse-engineered sequence diagrams.

The analysis was implemented as part of the RED toolkit for reverse engineering of sequence diagrams. The goal of the toolkit is to provide general, effective, and efficient reverse engineering of second-generation UML sequence diagrams for Java software. The control flow analysis completely solves one of the key challenges for the toolkit: the problem of mapping control-flow graphs to the UML control-flow primitives. The analysis implementation was evaluated experimentally on a large number of components. Our results confirmed its practicality and provided important insights into the structure of the reverse-engineered sequence diagrams.

## 1.3 Diagram Transformations

Our experiments with the algorithm revealed that the mapping from intra-method control flow to diagram elements sometimes may produce diagrams that are too verbose. To address this problem, we have defined several *transformations of sequence diagrams*. These transformations make the reverse-engineered diagrams easier to comprehend, while at the same time preserving their meaning. As a result, tools that employ our analysis can produce more compact diagrams which are more useful for tool users. Our extensive experiments show that the proposed transformations are very effective in simplifying the diagrams.

## 1.4 Test Coverage Tool

Based on the control flow analysis, we have built a prototype *coverage tool* for testing based on reverse-engineered sequence diagrams. The tool supports a testing approach proposed in [3] and also provides functionality needed for similar testing techniques [4, 28]. The Round-trip Scenario Test approach from [3] requires coverage of all possible control-flow decisions in sequence diagrams during testing of object interactions. The output of our control flow analysis provides a natural basis for a coverage tool for such testing. We present preliminary results from applying the tool to the Mauve open-source test suite for the standard Java libraries. Our study determined how well the Mauve tests cover different control-flow aspects of object interactions, and provided insights about potential weaknesses of the tests.

## 1.5 Contributions

The contributions of this work are:

- the first general algorithm for mapping intra-method flow of control to second-generation UML
- several transformations for improving the structure of the reverse-engineered sequence diagrams
- extensive experimental evaluation of the control flow analysis and the diagram transformations
- a test coverage tool based on the diagrams and a preliminary study on a real-world test suite

```
public abstract class NumberFormat {
   public int maxDigits() { ... }
   public int minDigits() { ... }
}
public class FormatSymbols {
   public char sep() { ... } // separator
}
public class DecimalFormat extends NumberFormat {
   ...
   public String toPattern(boolean local) { ... }
   private String s1, s2;
   private FormatSymbols sym;
   private static final char SEP = ',';
}
```

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

# 2. BACKGROUND

In this section we discuss the control-flow primitives of UML and describe the role of the control flow analysis in the reverse-engineering toolkit.

## 2.1 Toolkit for Reverse Engineering

The control flow analysis is designed and implemented as part of the ongoing work on the RED toolkit. Given a set of Java classes, a RED user can choose any method `m` from these classes and can generate a sequence diagram that represents the object interactions triggered by an invocation of `m`. To solve this problem, the toolkit includes several static analyses—for example, call graph construction [21], call chain analysis [20], and the control flow analysis described in this paper.

For each method that is shown in the sequence diagram, the control flow analysis examines the control-flow graph (CFG) of the method and creates a method-level data structure that encodes the relevant aspects of the method's control-flow behavior. Subsequent display of the reverse-engineered diagram (currently being implemented) makes use of the data structures created for the individual methods. The analysis is specifically designed to operate on CFGs, which makes it language-independent—the only requirements for analyzing a method (or a procedure in a procedural language) are certain CFG properties that are described later. Furthermore, this approach allows reverse engineering even when the source code is not available.

**Example.** Consider the Java classes in Figure 1. These sample classes resemble several classes from the standard library package `java.text`. Suppose that the user provides these classes to RED and wants a sequence diagram for the interaction triggered by a call to method `toPattern`. Figure 2 shows the control-flow graph of the method and the reverse-engineered sequence diagram. For this example we assume that the user is not interested in the calls made by the methods invoked by `toPattern`. The *loop*, *opt*, *break*, and *alt* elements represent the flow of control during the object interactions, as described shortly. These elements are based on the data structure produced by our control flow analysis of the CFG.

## 2.2 UML Sequence Diagrams

A sequence diagram contains objects and messages exchanged among these objects. In addition, the new generation of UML (version 2.0, to be finalized in April 2004)

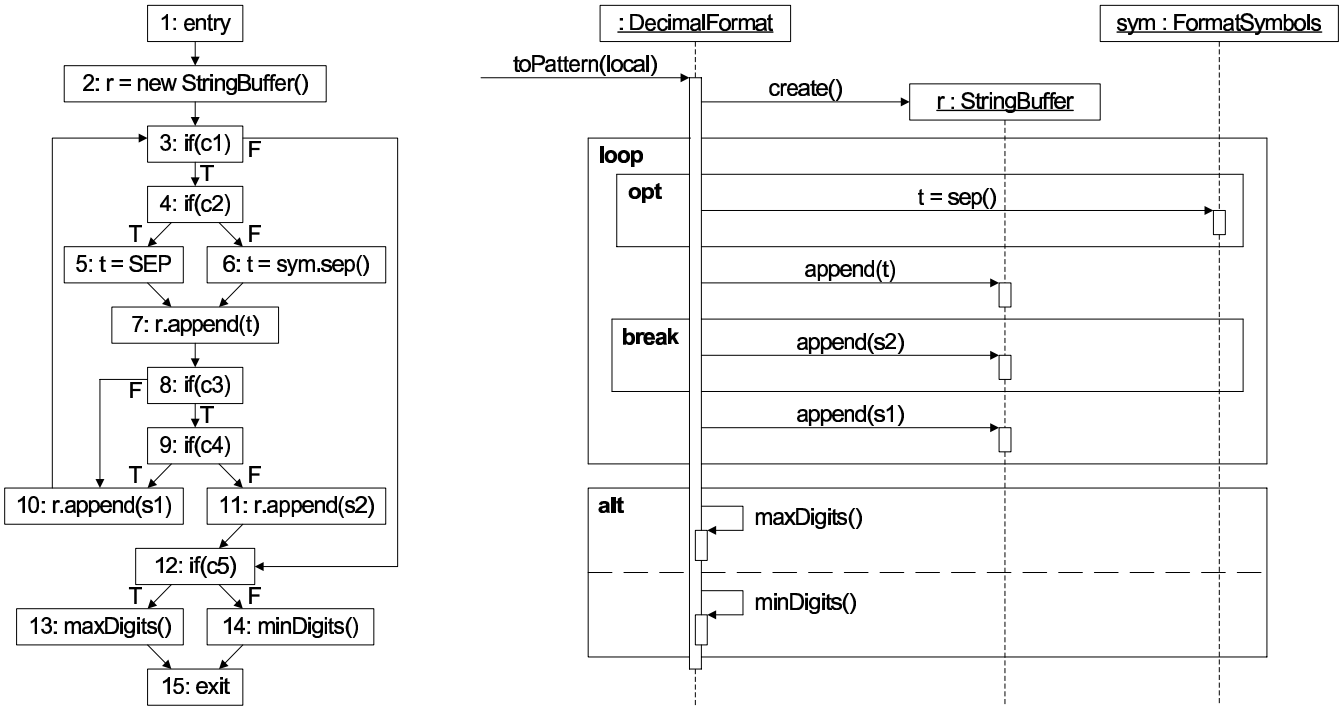



**Figure 2: (a) Control-flow graph for `toPattern` (b) Reverse-engineered sequence diagram for `toPattern`**

defines *interaction fragments* as diagram entities that represent various aspects of the interaction [16]. For the purposes of this work, four kinds of interaction fragments are of particular importance: *opt*, *alt*, *loop*, and *break* fragments. They provide the fundamental control-flow primitives that are used in the reverse-engineered sequence diagrams. Examples of these fragments are shown in Figure 2. A fifth kind of fragment—a *message* fragment—represents messages that are sent and received by objects.[1]

An opt, loop, or break fragment encloses an ordered sequence of other fragments. In Figure 2, the loop fragment encloses an ordered sequence containing four elements: an opt fragment, a message fragment `append(t)`, a break fragment, and a message fragment `append(s1)`. The opt fragment encloses a sequence containing a message fragment `t=sep()`, and the break fragment encloses a sequence containing a message fragment `append(s2)`. A message fragment does not enclose any other fragments.

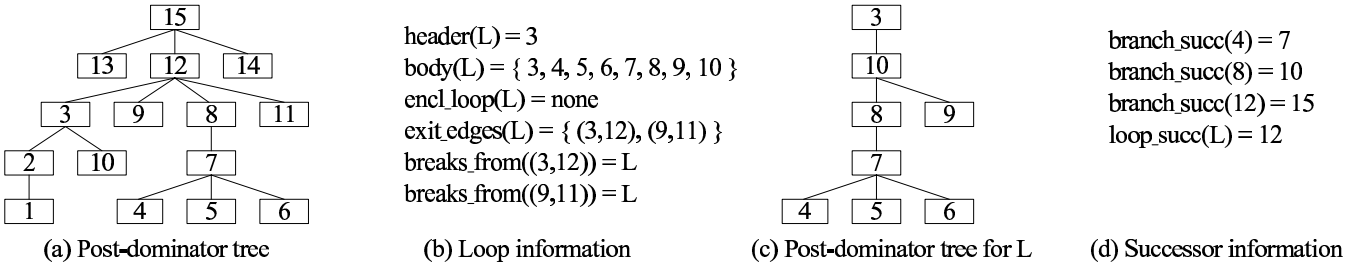
A sequence of fragments represents one or more sequences (traces) of run-time events [16]. An opt fragment describes optional behavior guarded by some condition. The sub-trace represented by the fragments inside an opt fragment is executed if the condition is true and skipped if the condition is false. For example, the opt fragment in Figure 2 is guarded by the condition `!c2` corresponding to CFG node 4. The enclosed message `t=sep()` is sent if and only if `c2` is false.

An alt fragment describes two or more mutually-exclusive alternatives in behavior. Each alternative is represented by a separate ordered sequence of fragments and is guarded by a particular condition. The set of traces defined by an alt fragment is the union of the sets of traces for the alternatives. For example, the alt fragment in Figure 2 has two alternatives. The first one is guarded by condition `c5` from CFG node 12 and encloses the sequence containing message fragment `maxDigits()`. The second one is guarded by `!c5` and encloses the sequence containing `minDigits()`.

The sequence of fragments enclosed in a loop fragment is repeated until the guard condition becomes false. For the loop in Figure 2, the sequence opt, `append(t)`, break, `append(s1)` is repeated until condition `c1` at CFG node 3 becomes false. The loop can also exit through the break fragment: if condition `c3&&!c4` associated with the break fragment ever becomes true, `append(s2)` is executed, the loop terminates, and the alt fragment is executed next. In general, a break fragment represents a "breaking" scenario: first the fragments inside the break are executed, and then the execution of the fragment enclosing the break completes immediately.

A break fragment, as defined by UML 2.0, breaks out of the immediately surrounding fragment. This definition is overly restrictive and makes it impossible for a reverse-engineering analysis to express the semantics of real-world code. We have adopted a generalized form of break fragment that allows breaking out of multiple enclosing fragments. Such a generalized break specifies the enclosing fragment out of which it is breaking. For example, if a break fragment $F_3$ is enclosed in $F_2$ which in turn is enclosed in $F_1$, $F_3$ could be of the form "break out of $F_1$". The UML notation can be easily augmented to represent this extension by labeling the corresponding enclosing fragment and using the label when displaying the break fragment; the examples in this paper use this approach.

[1]Strictly speaking, [16] treats the two endpoints of a message as separate fragments. This distinction is irrelevant for our work.




(a) Post-dominator tree (b) Loop information (c) Post-dominator tree for L (d) Successor information

**Figure 3: Post-dominators, loops, and successors.**

# 3. PHASE I: PREPROCESSING

The control flow analysis traverses the CFG of a method and maps different subgraphs to the interaction fragments described earlier. In Sections 3.3 and 3.4 we define the concepts of *branch successors* and *loop successors*. Branch and loop successors play a central role in the algorithm; they are computed before the mapping is performed. Sections 3.1 and 3.2 provide some standard background information needed to define these concepts.

## 3.1 Entry, Exit, and Post-Dominators

Consider a control-flow graph $G$. An *entry node* of $G$ is a node that does not have any predecessors, and an *exit node* of $G$ does not have any successors. We assume that the analyzed CFG has exactly one entry node. In Java an exit node corresponds to a `return` statement or a `throw` statement. In general, a CFG may have multiple exit nodes. The algorithm becomes more complicated in the presence of multiple exit nodes—for example, it becomes necessary to use information about control dependencies [8, 6] between CFG branch nodes and exit nodes. For brevity we discuss only the simpler case in which the CFG has a single exit node, and that node corresponds to a `return` statement. The general treatment of multiple exit nodes is described in [27]; our implementation fully handles this general case. We assume that each node in $G$ is reachable from the entry node and reaches the exit node.

Consider a control-flow graph $G$ with a single exit node that is reachable from all nodes. Node $n_2$ *post-dominates* $n_1$ if every path from $n_1$ to the exit node contains $n_2$. Node $n_2$ *immediately post-dominates* $n_1$ if $n_2$ post-dominates $n_1$ and any other post-dominator of $n_1$ is also a post-dominator of $n_2$. The immediate post-dominance relation can be represented by a post-dominance tree in which each parent node is the immediate post-dominator of its children. The root of the tree is the exit node. The post-dominance tree for the CFG from Figure 2 is shown in Figure 3a. Our implementation uses one of the algorithms from [14] to compute post-dominance trees.

## 3.2 Loops

In a *reducible* control-flow graph $G$, the edges can be partitioned into forward edges and back edges. The forward edges form an acyclic graph. Edge $(x, y)$ is a back edge iff $y$ is an ancestor of $x$ in the depth-first spanning tree rooted at the entry node. Intuitively, a reducible control-flow graph has "normal" loop structure. A loop in a reducible CFG is a strongly-connected subgraph $L$ such that [2]:

- Exactly one node $n \in L$ has an incoming edge $(n', n)$ such that $n' \notin L$. Node $n$ is the header node of $L$; we will denote it by $header(L)$.
- The set of nodes in $L$ is exactly the set of CFG nodes that are reachable from $header(L)$ and reach some $n'$ which is the source of a back edge $(n', header(L))$. We will denote this set of nodes by $body(L)$.

A node can be the header node for at most one loop. For any two loops $L_1$ and $L_2$, sets $body(L_1)$ and $body(L_2)$ are either disjoint or one is a proper subset of the other. The loop structure of a reducible CFG can be determined using Tarjan's interval-finding algorithm [24]; our current implementation uses a somewhat simpler approach from [2]. The control flow analysis determines all loops $L_i$ together with $header(L_i)$ and $body(L_i)$, and computes some additional information related to these loops:

- Enclosing loop $encl\_loop(L_i)$: the smallest $L_j \neq L_i$ such that $body(L_i) \subset body(L_j)$. If $L_i$ is a top-level loop, $encl\_loop(L_i) = none$.
- For each CFG node $n$: $encl\_loop(n)$ is the smallest $L_i$ such that $n \in body(L_i)$. In case $n$ is not inside any loop, $encl\_loop(n) = none$.

The control-flow features of Java ensure that the CFG of a method is reducible. Even if irreducibility is possible (e.g., in C and C++ due to `goto`), in reality it is rarely observed. Thus, we chose to design the control flow analysis under the assumption of reducibility; our implementation verifies this assumption for each input CFG. Various approaches for defining and identifying loops in irreducible graphs [17] could potentially be used to generalize our analysis; we plan to investigate this problem in the future.

## 3.3 Branch Successors

A CFG node is a *branch node* if it has at least two outgoing edges. For some of these nodes the analysis creates alt fragments. For this, it is necessary to determine which fragment should follow the alt in the enclosing fragment sequence, and which CFG node should be the starting point when constructing this next fragment. For example, for node 4 in Figure 2, the analysis will create an alt fragment which subsequently will be transformed into an opt fragment. The fragment following the alt in the loop's fragment sequence will be constructed starting from node 7, which is the "merge point" of the two branches coming out of 4. For a branch node $n$, the *branch successor* of $n$ is the node from which the analysis must continue after completing the processing of the fragment created for $n$. We denote this node by $branch\_succ(n)$. Figure 3d lists the branch successors for branch nodes 4, 8, and 12 from Figure 2.

Consider a branch node $n$ with outgoing edges $(n, n_i)$. If $\mathit{encl\_loop}(n) = \mathit{none}$, the branch successor is defined to be the lowest common ancestor of all $n_i$ in the post-dominator tree for the CFG. For example, for node 12 in Figure 2, the lowest common ancestor of 13 and 14 in the tree from Figure 3a is 15; therefore 15 is the branch successor for 12. In general, a common ancestor in the tree represents a merge point for all branches coming out of $n$. The lowest common ancestor is the merge point that is the "closest" to $n$, and it corresponds to the rest of the enclosing fragment sequence.

If $\mathit{encl\_loop}(n) \neq \mathit{none}$, the notion of a branch successor must be restricted to the flow of control that stays within the loop. For example, for node 8 in Figure 2, the lowest common ancestor of 9 and 10 in the tree from Figure 3a is 12, which is not even in the loop. Thus, the method-level post-dominator tree cannot be used to compute the branch successor.

To address this problem, we define the set $\mathit{exit\_edges}(L)$ of loop exit edges for a loop $L$ as $\{(n_1, n_2) \mid n_1 \in \mathit{body}(L) \wedge n_2 \notin \mathit{body}(L)\}$. Each such edge $e$ is associated with the loop out of which it is breaking; this is either $L$ or some loop in which $L$ is nested. More precisely, $\mathit{breaks\_from}(e)$ is the largest loop $L'$ such that $e$ is in $\mathit{exit\_edges}(L')$. For example, in Java a loop exit edge that crosses several nested loops may be due to a labeled `break` statement [11]. Figure 3b shows the loop exit edges for the loop from Figure 2a.

Our algorithm creates break fragments for loop exit edges. Thus, such edges must be ignored when determining branch successors for nodes in loops. We formalize this intuition by defining the notion of *post-dominance inside a loop*. Consider two nodes $n_1, n_2 \in \mathit{body}(L)$. Node $n_2$ post-dominates $n_1$ inside loop $L$ if every path from $n_1$ to $\mathit{header}(L)$ that stays entirely inside $L$ contains $n_2$. The post-dominance tree for a loop $L$ can be defined as the post-dominance tree for the subgraph with edge set $\{(n, m) \mid n, m \in \mathit{body}(L)\}$ in which $\mathit{header}(L)$ is designated as the subgraph's exit node. When $n_2$ post-dominates $n_1$ inside $L$, this means that if $n_1$ is reached during some iteration of $L$ and subsequently the iteration completes successfully—that is, $\mathit{header}(L)$ is eventually reached—then $n_2$ is reached after $n_1$ as part of that same iteration. Figure 3c shows the post-dominance tree for the loop from Figure 2a.

If $\mathit{encl\_loop}(n) = L$, we consider only edges $(n, n_i)$ that do not belong to $\mathit{exit\_edges}(L)$. If there are at least two such edges, we define the branch successor of $n$ to be lowest common ancestor of all such $n_i$ in the post-dominator tree for $L$. If less than two $n_i$ are in $L$, $n$ is not a branch node with respect to the loop-only control flow and there is no branch successor. This definition ensures that the branch successor of $n$ represents a merge point that is in the same loop as $n$ and belongs to the same iteration of that loop. For node 8 in Figure 2, we consider the post-dominator tree in Figure 3c and define the branch successor of 8 to be the lowest common ancestor of 9 and 10 in that tree—namely, node 10.

## 3.4 Loop Successors

Whenever the analysis encounters the header node of a loop, it creates a loop fragment. In this case, it is necessary to determine which fragment should follow the loop fragment in the enclosing fragment sequence, and from which CFG node the construction of this next fragment should start. For example, for node 3 in Figure 2, the analysis will create a loop fragment. The fragment following the loop in the top-level fragment sequence will be constructed starting from node 12. This is the merge point for the two possible ways to exit the loop, corresponding to the two loop exit edges $(3, 12)$ and $(9, 11)$.

For each loop $L$ we determine a *loop successor*, denoted by $\mathit{loop\_succ}(L)$. The definition is based on the loop exit edges for $L$. Let $\mathit{jump}(L)$ be the subset of exit edges $e$ for $L$ such that $\mathit{breaks\_from}(e) = L$. If $L$ is nested inside some other loop, this set contains all and only exit edges for $L$ that "jump" to $L$'s surrounding loop but not any further. Thus, we are interested *only* in exit edges that represent flow of control that follows $L$ in its surrounding loop. Any other exit edge represents a continuation of an iteration of some outer loop (not $L$'s surrounding loop), and therefore should be ignored when determining the successor of $L$.

The definition considers two cases. When $\mathit{encl\_loop}(L) = \mathit{none}$, the loop successor is the lowest common ancestor for all targets of edges from $\mathit{jump}(L)$ in the method-level post-dominator tree. For example, for the loop in Figure 2, these targets are nodes 11 and 12. In the tree from Figure 3a, their lowest common ancestor is 12; therefore, the loop successor is 12. Intuitively, this is the earliest common point for all possible executions after the loop terminates.

If $\mathit{encl\_loop}(L) = L'$, we consider the post-dominator tree for $L'$ and the lowest common ancestor in that tree for all targets of edges from $\mathit{jump}(L)$. As with branch successors, here we consider only flow of control that stays inside $L'$. In this case the loop successor represents the earliest merge point for all possible ways to exit $L$ and to continue with the current iteration of $L'$. If $\mathit{jump}(L) = \emptyset$, we define $\mathit{loop\_succ}(L) = \mathit{none}$; even though somewhat unusual, this case is possible and is handled by our algorithm.

# 4. PHASE II: FRAGMENT CONSTRUCTION

After computing the information described in the previous section, our analysis uses the algorithm from Figures 4 and 5 to construct a set of interaction fragments. The algorithm traverses the control-flow graph and creates fragments that correspond to the structure of that graph. For example, whenever the algorithm encounters an invocation expression, it creates a new message fragment and adds it to the fragment structure. Note that a polymorphic invocation expression may represent several possible messages that are being sent to different receiver objects. Since the focus of our work in the intra-method flow of control, we are not concerned with this issue and the algorithm maps the invocation expression to a single message fragment. Future analyses built on top of our analysis may represent the inter-method flow of control due to polymorphism in a different manner.

The algorithm is designed under two assumptions about the input control-flow graph. First, the graph should be reducible, as discussed in Section 3. Second, we assume that (1) a graph node corresponds to at most one invocation expression, and (2) a branch node does not correspond to an invocation expression. Conditions (1) and (2) can be trivially satisfied by introducing auxiliary variables and statements.

**input** Control-flow graph $G$ and all info from Section 3
**output** Fragment sequence $s$ constructed by *main*
**proc** *main*
[1] create empty fragment sequence $s$
[2] *processSequence*$(s, G.entry, G.exit)$
**proc** *processSequence*$(seq, start, stop)$
[3] $L :=$ *encl_loop*$(start)$
[4] $n :=$ *start*
[5] while $n \neq$ *stop* and $n \neq$ *none*
[6] if *encl_loop*$(n) \neq L$
[7] $n$ must be the header node of some loop $L'$
[8] *processLoop*$(seq, L')$
[9] $n :=$ *loop_succ*$(L')$
[10] if $n =$ *start* then $n :=$ *none*
[11] continue with the next iteration for [5]
[12] if $n$ contains a call
[13] append a new message fragment to *seq*
[14] *breaks* $:= \emptyset$
[15] if $L \neq$ *none*
[16] *breaks* $:= \{ m \mid (n, m) \in$ *exit_edges*$(L) \}$
[17] for each $m \in$ *breaks*
[18] *processBreak*$(seq, n, m)$
[19] *rest* $:= \{ m \mid (n, m) \in G \wedge m \notin$ *breaks*$\}$
[20] if *rest* $= \emptyset$ then *next* $:=$ *none*
[21] if *rest* $= \{ m \}$ then *next* $:= m$
[22] if *rest* $= \{ m_1, \ldots, m_k \}$ for $k > 1$
[23] *processAlt*$(seq, n,$ *rest*$)$
[24] *next* $:=$ *branch_succ*$(n)$
[25] $n :=$ *next*
[26] if $n =$ *start* then $n :=$ *none*

**Figure 4: Algorithm for fragment construction.**

Consider our running example. Given the control-flow graph for `toPattern`, the algorithm produces the fragments in Figure 6. Note that the fragments contain redundant information—for example, two of the alt fragments have empty alternatives. After some "cleanup" post-processing described later, we obtain the fragments shown in Figure 7.

Procedure *processSequence* in Figure 4 creates a sequence of fragments starting from some CFG node *start*. The graph is traversed until CFG node *stop* is encountered, or until there are no more nodes to process. For example, consider edge $(4, 6)$ in Figure 2. This edge corresponds to one alternative inside an alt fragment. To construct the fragment sequence for this alternative, *processSequence* will be invoked with *start* $= 6$ and *stop* $= 7$. In this case the construction of the sequence must stop at node 7, which is the branch successor of node 4.

As the algorithm encounters nodes during the traversal (lines 5–26), it "populates" the current sequence with the appropriate fragments. First, if the current node $n$ is enclosed in a loop $L'$ different from the enclosing loop $L$ for *start* (line 6), this means that $n$ is the header of $L'$ and $L'$ is enclosed in $L$. In this case *processLoop* creates the corresponding loop fragment, adds it to the current fragment sequence, and recursively builds the new fragment sequence inside the new loop fragment. The traversal then continues from the loop successor of $L'$.

For the CFG in Figure 2, *processSequence* will be invoked with *start* $= 1$ and *stop* $= 15$ in order to create the top-level fragment sequence. The traversal will first encounter the

**proc** *processLoop*$(seq, L)$
[27] append a new loop fragment $f$ to *seq*
[28] create an empty internal sequence $seq_2$ inside $f$
[29] *processSequence*$(seq_2,$ *header*$(L),$ *none*$)$
**proc** *processBreak*$(seq, n, m)$
[30] append a new break fragment $f$ to *seq*
[31] create an empty internal sequence $seq_2$ inside $f$
[32] $L :=$ *breaks_from*$((n, m))$
[33] *processSequence*$(seq_2, m,$ *loop_succ*$(L))$
**proc** *processAlt*$(seq, n,$ *rest*$)$
[34] append a new alt fragment to *seq*
[35] for each $m_i \in$ *rest*
[36] add a new alternative $a_i$ to the alt fragment
[37] create an empty internal sequence $seq_i$ inside $a_i$
[38] *processSequence*$(seq_i, m_i,$ *branch_succ*$(n))$

**Figure 5: Algorithm for fragment construction.**

constructor call when $n = 2$ and will create the corresponding message fragment (lines 12–13). For $n = 3$, *encl_loop*$(1)$ is different from *encl_loop*$(3)$ at line 6, and therefore a new loop fragment must be constructed and added to the top-level sequence. The loop successor is node 12 and the traversal will continue from that node. Due to nodes 12, 13, and 14, an alt fragment will be added to the top-level sequence (lines 22–24).

The creation of a fragment sequence enclosed inside a loop (line 29) does not use a stopping node. Rather, the traversal stops whenever the loop header is reached again along back edges. The checks at lines 10 and 26 are responsible for ensuring this termination.

At lines 15–18, the algorithm identifies outgoing edges from $n$ that correspond to break fragments. For each loop exit edge $(n, m)$ for $L$, a separate break fragment is added to the current sequence. For each one, a sequence is constructed recursively using as stopping node the loop successor of the loop from which $(n, m)$ breaks. Intuitively, this loop successor is the starting point of another fragment in some upper-level fragment sequence.

For the loop in Figure 2, *processSequence* will be invoked with *start* $= 3$ and *stop* $=$ *none* to create the sequence enclosed inside the loop fragment. Due to exit edge $(3, 12)$, a new break fragment is added as a first element of that sequence. For this new fragment, *processSequence* is called at line 33 with *start* $= 12$ and *stop* $=$ *loop_succ*$(L) = 12$; as a result, the sequence remains empty. Later, exit edge $(9, 11)$ results in another break fragment for which *processSequence* with *start* $= 11$ and *stop* $= 12$ adds a message fragment for `r.append(s2)` to the sequence inside this break fragment. In both cases the traversals stop at node 12, which is the starting point for the alt fragment in the top-level sequence.

Lines 19–26 process the remaining outgoing edges for $n$. If there are at least two such edges, an alt fragment is constructed by calling *processAlt*. For example, node 4 has two outgoing edges neither of which is a loop exit edge. A corresponding alt fragment is created and populated recursively. The stopping node for both alternatives is 7, which is the branch successor of 4. The traversals must stop at node 7 because it is the starting point for constructing the message fragment for `r.append(t)`.

Clearly, the algorithm may produce redundant information. For example, the first opt fragment in Figure 6 con-

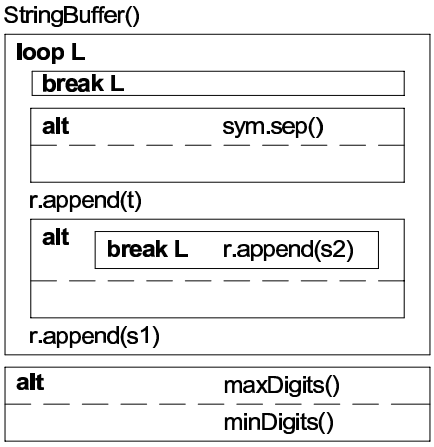



**Figure 6: Fragments constructed by the algorithm.**

tains an empty alternative corresponding to `t = SEP`. This alternative does not represent any useful information about messages and should be eliminated. Such redundancies are removed during "cleanup" post-processing. Figure 7 shows the cleaned-up version of the fragments from Figure 6. We use the rules described below, and apply them repeatedly until no more simplifications are possible. The rules are:

- If an alternative in an alt fragment $A$ has an empty enclosed fragment sequence, this alternative is removed. If after the removal there is only one remaining alternative, $A$ is replaced by an opt fragment.
- If an opt fragment $O$ encloses an empty fragment sequence, $O$ is removed.
- If a break fragment $B$ encloses an empty fragment sequence, $B$ is removed if (1) $B$ is the first or last element in the enclosed sequence of some loop fragment, and (2) $B$ crosses only one level of loop nesting.
- If a loop fragment $L$ encloses an empty fragment sequence, $L$ is removed.

The conditions for the third rule ensure that the removed break fragment does no contain any useful information. For example, if a break fragment has an empty sequence but appears in between two messages inside a loop, it will not be removed because it represents an "abrupt" exit from the loop. If a break fragment satisfies the conditions in the third rule, it need not be represented in the diagram because it is implied by the "normal" exit of the surrounding loop fragment.

# 5. PHASE III: TRANSFORMATIONS

The approach presented in the previous section can produce a set of fragments for an arbitrary reducible control-flow graph. The structure of the resulting fragments can be improved by using several *fragment transformations*. The goal of these transformations is to simplify the fragment structure without altering its meaning. The motivation for employing such transformations is to make the reverse-engineered sequence diagrams easier to comprehend. Our experience examining the output of the analysis indicates that deep nesting makes it harder to understand and to display

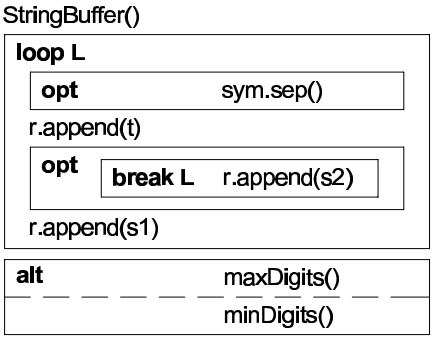



**Figure 7: Fragments after phase II.**

the diagrams. Thus, we define transformations that reduce nesting depth. Section 7 describes experimental results that demonstrate the effectiveness of these transformations.

The basic idea of our techniques is to move a nested fragment one level up in the fragment nesting structure. The first category of such transformations can be applied to an alternative in an alt fragment. Suppose that the fragment sequence for an alternative contains a single opt fragment. In this case the condition guarding the alternative and the condition guarding the opt can be combined. The sequence inside the opt fragment can be "lifted" one level up to become the sequence for the alternative. This eliminates the nested opt fragment. Consider the last alt fragment in Figure 7, with the corresponding condition `c5` from Figure 2. For the sake of illustration, suppose that the first alternative contained only an opt fragment surrounding `maxDigits()`, and this opt fragment corresponded to some condition `c6`. We can eliminate the opt fragment by moving `maxDigits()` into the alternative and by changing the condition associated with this alternative to become `c5 && c6`.

A similar transformation can be performed when an alternative in an alt fragment contains only an alt fragment. In this case each alternative of the inner alt can become an individual alternative of the outer alt, and the inner alt can be eliminated. Again, this requires combining of the condition for the original alternative in the outer alt with the individual conditions for the alternatives in the inner alt.

A second category of transformations considers an opt fragment whose enclosed sequence contains *only* alt, opt, and break fragments. Since each of the enclosed fragments has a guarding condition, we can eliminate the outer opt fragment and move the inner fragments one level up. Of course, the guarding conditions of these inner fragments have to be augmented with the condition of the removed opt fragment. The second opt fragment in Figure 7 illustrates this case. We can eliminate this fragment and replace it with the inner break fragment. The guarding condition of the break fragment changes from `!c4` to `c3 &&!c4`. After this transformation, we obtain the final fragment structure which is used in the sequence diagram from Figure 2.

This technique can be generalized for the case when the opt's sequence contains not only alt/opt/break fragments, but also a *single contiguous subsequence* with only message fragments and loop fragments in it. This subsequence can be used as the internal sequence of a new opt fragment which has the same condition as the old one. The rest of the el-

ements (alt, opt, and break) are moved one level up. For example, suppose that the second opt fragment from Figure 7 contained inside it the same break fragment followed by two message fragment $m_1$ and $m_2$. We can replace the outer opt with two new fragments: a break fragment with the appropriate modified condition, followed by a new opt fragment containing $m_1$ and $m_2$. This allows us to eliminate the nesting for the break fragment (but not for $m_1$ and $m_2$). Note that this could be done even if there were *multiple* non-adjacent subsequences of message/loop fragments inside the outer opt. However, in this case each such subsequence needs its own new opt fragment, which increases the total number of fragments and therefore may make the diagrams harder to comprehend. If there is only a single subsequence, the number of fragments does not change.

We are currently working on the design and implementation of several additional categories of transformations. However, as indicated by the results in Section 7, the two categories from above already have very beneficial effects.

# 6. TEST COVERAGE TOOL

Object-oriented software is heavily based on object interactions and *interaction testing* is essential in this context. Existing work [3, 1, 4, 10, 28] defines several techniques for interaction testing based on sequence diagrams or collaboration diagrams. These approaches can be applied naturally to reverse-engineered sequence diagrams, and the required coverage of different diagram elements can be measured automatically by instrumenting the code from which the diagrams were extracted.

Binder [3] considers testing of different beginning-to-end scenarios within a sequence diagram. His Round-Trip Scenario test pattern requires testing that exercises conditional and iterative behavior in the diagram. These testing requirements can be stated as coverage criteria for *transitions between fragments*. For example, for an opt fragment, it is necessary to exercise both the "true" behavior of entering the fragment sequence inside the opt, and the "false" behavior of skipping that sequence and continuing with the fragment that follows the opt. The behavior represented by alt and break fragments can be treated similarly. Thus, if it were possible to instrument the code and measure the coverage of these transitions, this would enable tool support for the Round-Trip Scenario approach. Furthermore, thorough coverage of control-flow decisions during object interactions is a necessary prerequisite for the approaches from [4, 28].

We have built a tool that allows coverage tracking for the intra-method conditional behavior in a reverse-engineered sequence diagram. The tool identifies a set of CFG edges that correspond to conditions in the diagram. These edges are instrumented and their coverage is observed at run time during the execution of the given tests. Since we want to distinguish the individual conditions that guard the fragments, the coverage tool considers the sequence diagram after phase II, before the transformations from phase III. For example, the second opt fragment in Figure 7 corresponds to CFG edges $(8, 9)$ and $(8, 10)$, and these two edges are tracked at run time. Other conditional behavior in Figure 7 is treated similarly. We also take into account the loop exit which was represented by a break fragment that was removed during cleanup post-processing. In general, each conditional behavior inside the reverse-engineered diagram is associated with CFG edges, and the coverage of these edges is measured at run time. We plan to represent the coverage information visually by displaying the sequence diagram and highlighting the conditional behavior that was not exercised. This will provide high-level view of those aspects of object interactions that may need additional testing. More details about the tool are described in [18].

| Component | Methods | Time (s) | (a) | (b) | (c) |
|---|---|---|---|---|---|
| `collator` | 157 | 4.84 | 56.1% | 17.8% | 26.1% |
| `date` | 136 | 5.43 | 82.4% | 5.1% | 12.5% |
| `decimal` | 136 | 0.77 | 81.6% | 6.6% | 11.8% |
| `message` | 176 | 1.33 | 77.3% | 5.7% | 17.0% |
| `boundaries` | 74 | 0.54 | 81.1% | 13.5% | 5.4% |
| `gzip` | 41 | 0.21 | 68.3% | 17.1% | 14.6% |
| `zip` | 118 | 0.54 | 72.0% | 21.2% | 6.8% |
| `math` | 241 | 0.96 | 50.6% | 33.2% | 16.2% |
| `pdf` | 330 | 0.74 | 78.2% | 7.9% | 13.9% |
| `mindbright` | 488 | 2.08 | 69.0% | 19.7% | 11.3% |
| `sql` | 60 | 0.32 | 63.3% | 16.7% | 20.0% |
| `html` | 298 | 1.42 | 62.4% | 18.5% | 19.1% |
| `jess` | 627 | 2.83 | 69.9% | 8.4% | 21.7% |
| `io` | 86 | 0.34 | 74.4% | 10.5% | 15.1% |
| `jflex` | 313 | 14.65 | 52.7% | 21.7% | 25.6% |
| `bytecode` | 625 | 6.65 | 60.2% | 19.2% | 20.6% |

**Table 1: Subject components.**

# 7. EMPIRICAL STUDY

This section summarizes some of the experimental results from our evaluation of the control flow analysis; more details about these and other results are available in [27]. We have implemented the analysis as part of RED, using the Soot framework [26]. The subject components used in the study are listed in Table 1. The components come from a variety of domains and typically represent parts of reusable libraries. The second column in the table shows the total number of non-abstract methods in each component.

The third column shows the running time of the analysis, in seconds. This is the total time to run all phases for all methods in a component, on a 900 MHz Sun Fire 280-R machine. The results strongly suggest that the cost of the analysis is practical. Certain aspects of our current implementation are somewhat inefficient, and we anticipate to achieve even lower running times in the near future.

For each method, we used phases I and II of the analysis to construct a set of fragments. We then considered all non-message fragments: these are fragments that would have to be represented by rectangles in a displayed sequence diagram. Our experience with some of the components suggests that when a diagram contains significant nesting of non-message fragments, it becomes hard to read and understand. For convenience, in the rest of this section we will use "fragments" to refer to non-message fragments.

We classified each method into one of three disjoint categories: (a) methods for which no fragments are created, such as simple get/set methods, (b) methods that do not have fragment nesting, and (c) methods with fragment nesting. The last three columns in Table 1 show the sizes of these categories, relative to the total number of methods. The majority of methods do not require any fragments, which is not surprising given the typical object-oriented program-

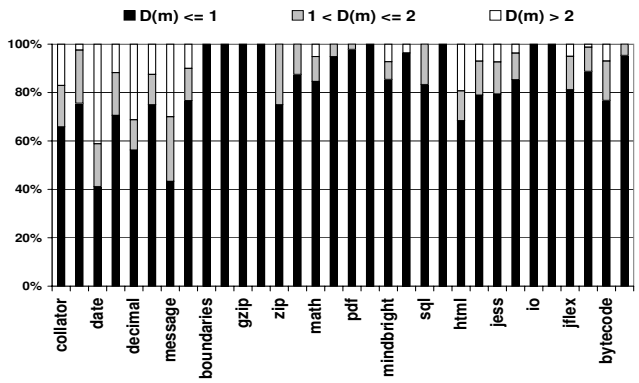



**Figure 8: Changes in average nesting depth.**

ming style. Still, fragments exist for a substantial number of methods: typically, at least 20% of the methods, and in some cases as many as 50%. Furthermore, many of these methods exhibit nesting of fragments. We will denote the set of methods in the third category by *Nested*.

Using the techniques from phase III, we transformed all methods in *Nested*. For each method $m$ we computed the average nesting depth $D(m)$ of the fragments in this method. The *nesting depth* of a fragment was defined as the number of its enclosing fragments. The methods from *Nested* were then separated into three sets: methods with $D(m) \leq 1$, methods with $1 < D(m) \leq 2$, and methods with $D(m) > 2$. The larger values of $D(m)$ indicate that the method's fragment structure is more complicated. For each component we computed this partitioning before and after the transformations from phase III. Figure 8 shows these measurements: for each component, there are two adjacent bars showing the distribution before and after phase III. For example, for `collator`, the first two bars in the figure show that the number of methods from *Nested* with $D(m) > 2$ decreases to less than 5%, while the number of methods with $D(m) \leq 1$ increases to about 75%.

The results indicate that the techniques from Section 5 successfully reduce unnecessary nesting of fragments. For example, from the 13 components in which there is room for improvement in category $D(m) \leq 1$, 10 components exhibit increase for this category by more than 10 percentage points. In two cases (`date` and `message`) this increase is as large as 30 percentage points. Similarly, the category for $D(m) > 2$ decreases in size, in several cases by more than 10 percentage points. The effect of these improvements on the comprehension of the diagrams will have to be evaluated eventually by users of Red. The anecdotal evidence from our experience strongly suggests that the reduction in nesting corresponds directly to the ease of understanding of the diagrams. This is also confirmed by the reduction of the total number of nested fragments (i.e., fragments with positive nesting depth) in methods from *Nested*. This improvement can be summarized as follows: for 14 out of the 16 components, the number of nested fragments is reduced by more than 10%; for 11 components the reduction is more than 20%; for 7 components the reduction is more than 30%; and for 3 components the reduction is more than 40%.

In addition to the evaluation of the static control flow analysis, we also present preliminary results from the dynamic analysis of run-time coverage of fragment transitions. Recall from Section 6 that the dynamic analysis is used in a test coverage tool to determine the coverage of intra-method control flow in the diagrams, based on Binder's Round-Trip Scenario testing approach [3]. We used the tool to evaluate several tests from the Mauve open-source test suite for the standard Java libraries (`sources.redhat.com/mauve`).

| Component | (a) Diagram-related | | (b) All branches | |
|---|---|---|---|---|
| | static | covered | static | covered |
| `collator` | 285 | 170 (59.6%) | 375 | 204 (54.4%) |
| `date` | 93 | 55 (59.1%) | 217 | 114 (52.5%) |
| `decimal` | 335 | 201 (60.0%) | 551 | 327 (59.3%) |
| `message` | 384 | 186 (48.4%) | 587 | 284 (48.4%) |
| `boundaries` | 26 | 24 (92.3%) | 28 | 26 (92.9%) |
| `gzip` | 44 | 30 (68.2%) | 60 | 37 (61.7%) |
| `zip` | 19 | 13 (68.4%) | 35 | 19 (54.3%) |

**Table 2: Coverage achieved by the Mauve tests.**

The experiments considered each method that was executed at least once by the tests and for which there was created at least one fragment. We then determined the set of all CFG branches in such methods—that is, all edges $(n, m)$ such that $n$ has at least two outgoing edges. Some of these branches correspond to transitions in the sequence diagrams. Part (a) of Table 2 shows the number of such diagram-related branches and their run-time coverage. The coverage results indicate potential weaknesses in some of the Mauve tests, in particular for `message`, `collator`, `date`, and `decimal`. We are currently investigating the reasons for this low coverage. Furthermore, additional tests will be created to achieve higher coverage, with the goal of contributing them to the Mauve project. We are also considering ways to incorporate this coverage information in visual displays of the diagrams.

Part (b) of Table 2 shows the total number of *all* CFG branches in the same set of methods, and their run-time coverage. There appears to be a strong correlation between the coverage of fragment-level flow of control and the traditional lower-level CFG branch coverage; additional results presented in [18] also support this observation. We plan to investigate this relationship in the near future. If in fact there exists strong correlation between the two, this would imply that testing of object interactions based on sequence diagrams—which is a natural level of abstraction for object-oriented software—also achieves high CFG branch coverage.

# 8. RELATED WORK

Several existing techniques employ dynamic analysis of run-time program behavior to perform reverse engineering of sequence diagrams or similar representations [23, 19, 7, 15, 5]. This approach has several drawbacks. First, the quality of the results depends on the particular execution that was observed, and on the input data for the execution. In many cases such input data is not available, especially for incomplete systems (e.g., reusable modules) that cannot be executed in a stand-alone manner. Even if input data is available, it is not possible to know how well the execution covers all possible aspects of the interaction. For example, it is not possible to have high confidence in the consistency between design and code, if this consistency is judged from sequence diagrams that were constructed from

execution traces. Similarly, for reengineering tasks, the incomplete run-time information may mislead the programmer into performing incorrect code modifications. For the same reason, sequence diagrams produced with dynamic analysis cannot be used for evaluating the adequacy of testing. Some dynamic reverse-engineering analyses completely ignore conditions and iterations [19, 7, 15]. Even for approaches that attempt to take them into account [23, 5], the quality of the results depends on pattern matching heuristics.

Reverse engineering of sequence diagrams through static analysis avoids these problems. Unfortunately, there is little existing work on this form of static analysis. The Together ControlCenter modeling tool (`borland.com/together`) includes such functionality as an advanced feature. Kollman and Gogolla [12] define a static analysis for reverse engineering of collaboration diagrams (conceptually similar to sequence diagrams). Tonella and Potrich [25] present a static analysis for reverse engineering of sequence diagrams and collaboration diagrams from C++ code. These approaches do not perform any form of control flow analysis and do not attempt to take advantage of the expressive power of the new generation of UML.

# 9. CONCLUSIONS AND FUTURE WORK

This work describes the first algorithm for mapping reducible control-flow graphs to UML interaction fragments, together with effective techniques for fragment simplification and a coverage tool for tracking the transitions between fragments. As part of the RED toolkit, the analysis solves one important problem for reverse engineering of sequence diagrams. We are currently examining additional fragment transformations to further simplify the diagrams. We also plan to extend the test coverage tool with a dynamic analysis that tracks the coverage of beginning-to-end paths in a sequence diagram, as proposed by several testing approaches.

# 10. REFERENCES


[1] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *International Conference on the Unified Modeling Language*, pages 383–395, 2000.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

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

[5] L. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *Working Conference on Reverse Engineering*, pages 57–66, 2003.

[6] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[7] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualising the execution of Java programs. In S. Diehl, editor, *Software Visualization*, LNCS 2269, pages 151–162, 2002.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Programming Languages and Systems*, 9(3):319–349, 1987.

[9] M. Fowler. *UML Distilled*. 2nd edition, 2000.

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

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

[12] R. Kollman and M. Gogolla. Capturing dynamic program behavior with UML collaboration diagrams. In *European Conference on Software Maintenance and Reengineering*, pages 58–67, 2001.

[13] C. Larman. *Applying UML and Patterns*. 2nd edition, 2002.

[14] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flow graph. *ACM Trans. Programming Languages and Systems*, 1(1):121–141, July 1979.

[15] R. Oechsle and T. Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In S. Diehl, editor, *Software Visualization*, LNCS 2269, pages 176–190, 2002.

[16] OMG. *UML 2.0 Infrastructure Specification*. Object Management Group, `www.omg.org`, Sept. 2003.

[17] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Trans. Programming Languages and Systems*, 24(5):455–490, Sept. 2002.

[18] M. Reddoch. Intra-method test coverage for reverse-engineered sequence diagrams. Master's thesis, Ohio State University, Mar. 2004.

[19] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Int. Conf. Software Maintenance*, pages 34–43, 2002.

[20] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in Java. In *International Symposium on Software Testing and Analysis*, July 2004. To appear.

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

[22] J. Rumbaugh, I. Jacobson, and G. Booch. *UML Reference Manual*. Addison-Wesley, 1999.

[23] T. Systä, K. Koskimies, and H. Muller. Shimba—an environment for reverse engineering Java software systems. *Software–Practice and Experience*, 31(4):371–394, Apr. 2001.

[24] R. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

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

[26] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[27] O. Volgin. Analysis of flow of control for reverse engineering of sequence diagrams. Master's thesis, Ohio State University, June 2004.

[28] Y. Wu, M.-H. Chen, and J. Offutt. UML-based integration testing for component-based software. In *International Conference on COTS-Based Software Systems*, 2003.