Coverage Criteria for Testing of Object Interactions in Sequence Diagrams

Atanas Rountev, Scott Kagan, and Jason Sawin

Ohio State University
{rountev,kagan,sawin}@cse.ohio-state.edu

Abstract. This work defines several control-flow coverage criteria for testing the interactions among a set of collaborating objects. The criteria are based on UML sequence diagrams that are reverse-engineered from the code under test. The sequences of messages in the diagrams are used to define the coverage goals for the family of criteria, in a manner that generalizes traditional testing techniques such as branch coverage and path coverage. We also describe a run-time analysis that gathers coverage measurements for each criterion. To compare the criteria, we propose an approach that estimates the testing effort required to satisfy each criterion, using analysis of the complexity of the underlying sequence diagrams. The criteria were investigated experimentally on a set of realistic Java components. The results of this study compare different approaches for testing of object interactions and provide insights for testers and for builders of test coverage tools.

1 Introduction

Object-oriented software presents a variety of new challenges for testing, compared to testing for procedural software [1]. For example, programs contain complex interactions among sets of collaborating objects from different classes. It is not sufficient to test a class in isolation—testing the interactions between instances of different classes is of critical importance [2, 1, 3]. A variety of techniques can be employed to test different aspects of object interactions. Several existing approaches for such testing [3–7] are based on *UML interaction diagrams*. UML defines two kinds of semantically-equivalent interaction diagrams: sequence diagrams and collaboration diagrams [8, 9]. In this paper we discuss only sequence diagrams; Figure 1a contains an example of such a diagram.

A sequence diagram shows the messages that are exchanged among several objects, as well as other control-flow information (e.g., if-then conditions that guard messages). Such diagrams capture important aspects of object interactions, and can be naturally used to define testing goals that must be achieved during testing. The testing requirements are related to certain elements of the diagrams. For example, it may be required to exercise all relationships of the form "object `X` send message `m` to object `Y`". More aggressive approaches consider not only individual messages, but also *sequences of messages*—for example,

all possible start-to-end message sequences in a diagram. Section 2 discusses in detail the previous work that proposes such approaches.

With the help of reverse-engineering tools, sequence diagrams can be extracted from existing code. Design recovery through reverse engineering is necessary during iterative development [10] and for evolving systems in which the design documents have not been updated to reflect code changes. Commercial tools already provide some functionality for such reverse engineering, both for class diagrams and for sequence diagrams. In addition, several static analyses proposed in the literature have considered various aspects of reverse engineering of sequence diagrams [11–14]. Reverse-engineered sequence diagrams are a natural source of program-based coverage criteria for testing of object interactions. If a reverse-engineering tool is used to construct a sequence diagram, a coverage tool can use this diagram as a basis for defining and measuring of coverage metrics during subsequent testing. Such a diagram reflects precisely the up-to-date state of the code, and therefore can be used for early and frequent testing.

The first goal of our work is to define *a family of coverage criteria for object interactions* based on reverse-engineered sequence diagrams. The criteria are generalizations of traditional control-flow criteria such as branch coverage and path coverage, and are defined in terms of the sequences of messages exchanged among a set of collaborating objects. Some of these criteria have appeared in previous work. However, there have been no attempts to define a unifying framework for such criteria and to use it for systematic investigation and comparison of different techniques for testing of object interactions. The work presented in this paper defines such a framework. At the center of the proposed approach is a data structure which we refer to as *interprocedural restricted control-flow graph* (IRCFG). This data structure represents in a compact manner the set of message sequences in a sequence diagram, and can be easily constructed as part of the reverse engineering of such a diagram. The IRCFG allows us to define systematically the family of test coverage criteria.

Our second goal is to design *a run-time analysis* based on the IRCFG. The run-time analysis observes the behavior of the code while tests are being executed, and gathers coverage measurements with respect to each criterion. Automated coverage measurements are essential for any program-based coverage criterion, and the run-time analysis is an important complement to the criteria.

The third goal of this work is to perform a comparison of the different criteria. We aim to obtain an estimate of the effort required to achieve high coverage for each criterion, and to compare these estimates. For each criterion c, we propose an approach which determines a lower bound p_c on the number of start-to-end IRCFG paths that guarantee the highest possible coverage for c. If for a given sequence diagram the value of p_c is very high, this indicates that the effort required to achieve high coverage for c may be prohibitive, and therefore weaker criteria should be used. Having such estimates provides valuable insights about the differences between the criteria, which in turn could allow better planning and management of the testing process.

The fourth goal of the work is to perform an experimental study that determines the values of p_c for different criteria on a set of realistic software components. Our experiments use 18 components from various Java libraries. The comparison of p_c across a diverse set of components provides insights into the inherent relationships between the different coverage criteria, and into the effort required to achieve high coverage for these criteria.

2 Testing and Sequence Diagrams

Several testing approaches proposed in the literature consider testing of object interactions based on sequence diagrams (or the semantically-equivalent collaboration diagrams). Binder [3] considers the set of all start-to-end paths in a sequence diagram, and defines a criterion for choosing a subset of paths to be covered during testing. The criterion requires coverage that is similar to traditional branch coverage: each decision outcome within the diagram must be covered by at least one start-to-end path. For example, if a message is sent under some condition c, the set of test cases should ensure that at least one path covers the case when c is true, and at least one path covers the case when c is false. We will refer to this criterion as the all-branches criterion; a precise definition of this approach is presented later in the paper.

Consider the sequence diagram in Figure 1a. This diagram represents the set of possible behaviors when message `m1` is sent to object `a`. Conditions `c1`, `c2`, and `c3` guard certain messages: for example, `m6` is sent to `b` only if `c3` is true. A start-to-end path in the diagram can be represented by the temporal sequence of messages that are exchanged between objects. For example, one such path is (`m1`,`m2`,`m4`,`m6`,`m2`,`m3`,`m4`). To satisfy the all-branches criterion, testing must execute enough start-to-end paths to cover all conditional behavior. One possible set of paths that satisfies this requirement is $p_1 = (\texttt{m1}, \texttt{m2}, \texttt{m3}, \texttt{m4}, \texttt{m5})$, $p_2 = (\texttt{m1}, \texttt{m2}, \texttt{m4}, \texttt{m6}, \texttt{m2}, \texttt{m3}, \texttt{m4})$, and $p_3 = (\texttt{m1}, \texttt{m2}, \texttt{m4}, \texttt{m6}, \texttt{m2}, \texttt{m4}, \texttt{m5})$.

Other testing approaches consider not only individual messages and their guarding conditions, but also entire sequences of messages. Jorgensen and Erickson [15] consider testing that exercises method-message paths and atomic system functions. A method-message path is a sequence of events of the form "method m_1 invokes method m_2; during this invocation, m_2 invokes m_3; during this invocation, m_3 invokes method $m_4 \ldots$". For example, in Figure 1a, the left-to-right sequence of messages (`m1`,`m6`,`m2`,`m3`) corresponds to a message-method path. In the subsequent discussion, we will use the more common term *call chain* to refer to such a sequence. An atomic system function, as defined in [15], is equivalent to the set of all start-to-end message sequences in a sequence diagram.

Abdurazik and Offutt [4] consider collaboration diagrams created during design, and define an approach for static checking and testing of the interactions among the diagram objects. Their technique requires coverage of start-to-end sequences of messages in the diagrams. Basanieri and Bertolino [16] define a testing approach that considers all message sequences in a sequence diagram and applies the category-partition method to choose the appropriate test data

for exercising these sequences. Fraikin and Leonhardt [6] describe the SeDiTeC tool for testing based on sequence diagrams. Their approach requires coverage of all possible sequences of messages in a set of related sequence diagrams. The diagrams are augmented with information about expected input and output values for method invocations, and these values are checked during test execution.

Briand and Labiche [5] consider functional system testing based on use cases and sequence diagrams (or collaboration diagrams) constructed during object-oriented analysis. Each scenario within a use case corresponds to a start-to-end path in the sequence diagram for that use case. They construct a regular expression that represents all start-to-end message sequences (i.e., all scenarios), and require coverage of all such sequences during testing. Wu et al. [7] propose an approach for testing of component-based software which uses UML collaboration/sequence diagrams and statecharts. One of the suggested techniques requires testing of all possible sequences of messages in a collaboration diagram.

3 Criteria for Reverse-Engineered Sequence Diagrams

The testing approaches discussed in the previous section are based on interaction diagrams that are constructed during analysis or design, before the corresponding implementation code is written. In general, there is no guarantee that design activities will produce a complete set of diagrams for all interactions in the system. An incomplete set of diagrams is a weak basis for comprehensive testing of object interactions. Another potential problem is that during code construction, the implementation often diverges from the original design. For example, in iterative development, tools for reverse engineering of design artifacts from the code are often necessary to make the design documents consistent with the actual implementation.

This paper considers sequence diagrams that are constructed automatically from existing code, using static analyses for reverse engineering [11–14]. Problems due to incomplete or outdated diagrams can be avoided with the use of reverse-engineered diagrams. Such diagrams can be constructed automatically from the latest version of the code, and for all relevant parts of the system. Furthermore, since the diagrams are created from the code, a coverage tool can easily determine what kinds of code instrumentation will be necessary in order to obtain run-time coverage metrics during test execution.

The approaches from Section 2 (with the exception of Binder's work [3]) have a common element: the requirement that all message sequences in an interaction diagram should be covered. This requirement is either used as a stand-alone coverage criterion, or as part of more general testing goals. When considered in the context of reverse-engineered sequence diagrams (rather than diagrams created during analysis or design), the requirement for all-paths coverage raises concerns similar to the ones from traditional CFG path coverage. Typically, CFG path coverage is considered to be infeasible in practice due to the potentially large number of paths. A similar question can be asked for testing of object interactions: is it practical to require coverage of all start-to-end paths in a

reverse-engineered sequence diagram? In fact, the reason Binder considers the weaker all-branches criterion is because, as he states, "the number of paths can easily reach astronomical numbers" [3].

This section presents a formal definition of three coverage criteria that are weaker versions of the all-paths criterion; one of them is the all-branches criterion. The criteria provide several options with different tradeoffs between testing effort and test comprehensiveness. Having such options is important in the presence of resource constraints for the testing process. Depending on these constraints, different criteria for systematic testing of object interactions can be employed. The criteria are generalizations of traditional control-flow-based criteria such as CFG branch coverage and CFG path coverage. We first define the notion of an *interprocedural restricted control-flow graph* (IRCFG), which can be thought of as the equivalent of a CFG for a sequence diagram. Figure 1b shows the IRCFG for the diagram from Figure 1a. Paths through the IRCFG correspond to sequences of messages in a sequence diagram. The proposed coverage criteria for object interactions are then defined formally based on the IRCFG.

3.1 Interprocedural Restricted Control-Flow Graph

An IRCFG contains a set of *restricted CFGs* (RCFGs), together with edges which connect these RCFGs. Each RCFG corresponds to a particular method and is similar to the CFG for that method, except that it is restricted to the flow of control that is relevant to message sending. In Figure 1b, each RCFG is shown within a rectangular box. For example, the top RCFG in the figure corresponds to method `m1`, which is invoked as a result of sending message `m1` to object `a`. A node in the RCFG for some method m represents a method invocation in the body of m. For example, the node labeled `m2` in the top RCFG in Figure 1b corresponds to some call to `m2` in the body of method `m1`. In the reverse-engineered diagram from Figure 1a, this call is represented by the message `m2` sent from `a` to `c`. The RCFGs also contain artificial nodes `start` and `end`. The start node represents the moment when the run-time execution enters the method, and the end node represents the moment when the flow of control returns back to the caller.

RCFG edges, shown with solid arrows in Figure 1b, represent the sequencing relationships between nodes. In Figure 1a, after the execution enters method `m1`, method `m2` is invoked. This is represented by the edge (`start`,`m2`) in the RCFG for `m1`. After this invocation of `m2` completes, either `m6` is invoked by `m1`, or `m1` completes without invoking `m6`. These two possibilities are represented by RCFG edges (`m2`,`m6`) and (`m2`,`end`) respectively. Sometimes we will refer to RCFG edges as *intraprocedural edges*. RCFGs are connected with *interprocedural edges*, shown in Figure 1b with dashed arrows. An interprocedural edge connects an RCFG node n with a start node that corresponds to some method that could be invoked by n. Note that due to polymorphism, there could be multiple interprocedural edges coming out of n. The interprocedural edges define a tree in which the nodes are RCFGs; we will refer to that tree as the *RCFG tree*.

Clearly, all information in the IRCFG is entirely based on the structure of the corresponding sequence diagram. Since we consider sequence diagrams that are

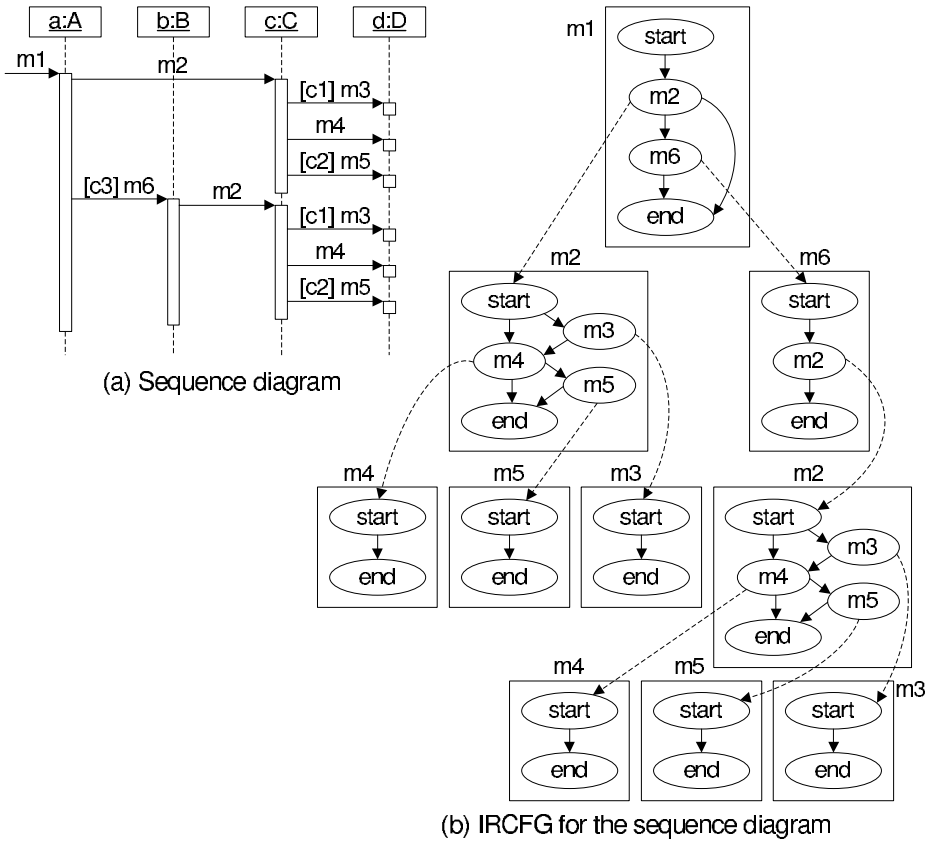

Fig. 1. Sample sequence diagram and the corresponding IRCFG.

constructed from existing code using some reverse-engineering static analysis, it should be straightforward to construct the IRCFG by augmenting the static analysis. Our implementation (described later) uses this approach: it extends an existing reverse-engineering analysis to construct the IRCFG.

3.2 Coverage Criteria

The IRCFG introduced in the previous section serves two purposes. First, it allows precise formal definition of coverage criteria for the corresponding sequence diagram. Second, it is the basis for a run-time analysis that measures the coverage achieved during testing. In this section we focus on the definition of the criteria; the run-time analysis is outlined in Section 4.

All-IRCFG-Paths Coverage The all-paths criterion, which we will refer to as *All-IRCFG-Paths*, requires coverage of the entire set of complete IRCFG paths.

Each complete path is a start-to-end traversal of the IRCFG. An example of such a path is

$$(\texttt{start}_{\texttt{m1}}, \texttt{m2}, \texttt{start}_{\texttt{m2}}, \texttt{m4}, \texttt{start}_{\texttt{m4}}, \texttt{end}_{\texttt{m4}}, \texttt{m5}, \texttt{start}_{\texttt{m5}}, \texttt{end}_{\texttt{m5}}, \texttt{end}_{\texttt{m2}}, \texttt{end}_{\texttt{m1}})$$

Let p be a sequence of RCFG nodes in which the first and the last node are `start` and `end` in the root RCFG, respectively. We will refer to p as a *complete IRCFG path* if it has the following property. Consider some node n_i in p, and let R be the enclosing RCFG for n_i. If the next node after n_i in the sequence p is node n_j, then one of the following must hold:

Case 1. If n_i is the start node of R, there must exist an intraprocedural RCFG edge (n_i, n_j) in R

Case 2. If n_i is not the start or the end node of R, then
- there exists an interprocedural edge (n_i, n_j), where n_j is the start node of some child of R in the RCFG tree, or
- there are no interprocedural edges starting at n_i, and (n_i, n_j) is an intraprocedural edge in R

Case 3. If n_i is the end node of R, then the parent of R in the RCFG tree contains an intraprocedural edge (n_k, n_j), and there is an interprocedural edge from n_k to the start node of R

The second alternative in Case 2 represents a situation when the body of the method invoked by n_i is not included in the diagram. For example, it is common to "stop" the reverse-engineered diagrams at library methods; in this case there is no interprocedural edge coming out of n_i.

It is important to note that not all complete IRCFG paths necessarily correspond to feasible run-time executions. Of course, this is a standard issue for any program-based criterion that uses some abstracted model of the tested code. For example, in traditional CFG path coverage, some CFG paths may be infeasible and complete coverage may not be possible. Even though it is impossible to completely eliminate infeasibility, there is a wide range of effective static analysis techniques that can reduce significantly the degree of infeasibility in program models such as CFGs and IRCFGs. For example, points-to analyses (e.g., [17]) can produce very precise calling-context-sensitive information about the calling relationships between methods, and branch correlation analysis (e.g., [18]) can identify certain classes of infeasible CFG paths. Static analyses for reverse engineering of sequence diagrams can employ such techniques to identify infeasible subpaths in the diagrams and in their corresponding IRCFGs. The investigation of this issue is beyond the scope of this paper, and the subsequent discussion assumes that all IRCFG paths are feasible.

An interesting question is how many complete IRCFG paths exist in a given IRCFG. Consider the example in Figure 1b. The invocation of `m2` from `m1` could lead to four distinct IRCFG subpaths. Similarly, the invocation of `m6` from `m1` may proceed along four distinct subpaths. Therefore, there are 16 complete IRCFG paths in which `m1` calls `m2` and `m6`. When we also consider the case in which `m6` is skipped, the total number of paths becomes 20. This example illustrates one fundamental concern with the *All-IRCFG-Paths* criterion: the number of paths could easily grow exponentially.

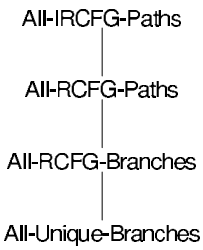

Fig. 2. Subsumption hierarchy for coverage criteria.

All-RCFG-Paths Coverage Next, consider all RCFG paths in an IRCFG. An *RCFG path* is a sequence of RCFG nodes within some RCFG R, beginning with the start node of R and finishing with the end node of R. Each pair of adjacent nodes in the path must correspond to an intraprocedural edge in R. For example, for the root RCFG in Figure 1b, there are two such paths: (`start`,`m2`,`end`) and (`start`,`m2`,`m6`,`end`). A complete IRCFG path could cover several RCFG paths. For example, consider again path

$$(\texttt{start}_{\texttt{m1}}, \texttt{m2}, \texttt{start}_{\texttt{m2}}, \texttt{m4}, \texttt{start}_{\texttt{m4}}, \texttt{end}_{\texttt{m4}}, \texttt{m5}, \texttt{start}_{\texttt{m5}}, \texttt{end}_{\texttt{m5}}, \texttt{end}_{\texttt{m2}}, \texttt{end}_{\texttt{m1}})$$

This complete path covers the following RCFG paths: $(\texttt{start}_{\texttt{m1}}, \texttt{m2}, \texttt{end}_{\texttt{m1}})$ in the root RCFG, $(\texttt{start}_{\texttt{m2}}, \texttt{m4}, \texttt{m5}, \texttt{end}_{\texttt{m2}})$ in the left child of the root, and the trivial `start-end` paths in the two leftmost leaves.

The *All-RCFG-Paths* criterion requires testing to exercise enough complete IRCFG paths to cover all RCFG paths. In Figure 1b, coverage for this criterion can be achieved with five (but not fewer) complete IRCFG paths. Coverage of all RCFG paths is similar to traditional CFG path coverage. Of course, unlike a CFG, an RCFG represents only a subset of the flow of control within a method (e.g., conditions that are irrelevant for calls are ignored). Furthermore, the criterion takes into account the calling context of a method. For example, for `m2` there are two RCFGs in Figure 1b—corresponding to call chains (`m1`,`m2`) and (`m1`,`m6`,`m2`)—and each start-to-end path in each of the two RCFGs should be covered.

All-RCFG-Branches Coverage Another potential source of exponential growth is the fact that the number of RCFG paths could be exponential in the size of the RCFG. We can eliminate this source by defining a criterion that requires coverage of all RCFG edges rather than all RCFG paths. This *All-RCFG-Branches* criterion is equivalent to Binder's approach discussed in Section 2. For our running example, the criterion can be satisfied with three complete IRCFG paths.

Unique RCFG Branches It is possible to define an additional simplification that leads to an even weaker (and easier to achieve) criterion. Consider the case when the tree contains several RCFGs for the same method, and each graph is

associated with different calling contexts for the corresponding method. If we require coverage of each RCFG edge regardless of the calling context, this defines a coverage criterion that is a simplified version of *All-RCFG-Branches*. In essence, we consider each unique RCFG edge regardless of how many times it occurs in the IRCFG, and require at least one occurrence to be covered by a complete IRCFG path. The new criterion will be denoted by *All-Unique-Branches*. For Figure 1b, this criterion can be satisfied by two complete IRCFG paths—for example,

$(\texttt{start}_{\texttt{m1}}, \texttt{m2}, \texttt{start}_{\texttt{m2}}, \texttt{m3}, \texttt{start}_{\texttt{m3}}, \texttt{end}_{\texttt{m3}}, \texttt{m4}, \texttt{start}_{\texttt{m4}}, \texttt{end}_{\texttt{m4}}, \texttt{m5}, \texttt{start}_{\texttt{m5}}, \texttt{end}_{\texttt{m5}}, \texttt{end}_{\texttt{m2}}, \texttt{end}_{\texttt{m1}})$ and $(\texttt{start}_{\texttt{m1}}, \texttt{m2}, \texttt{start}_{\texttt{m2}}, \texttt{m4}, \texttt{start}_{\texttt{m4}}, \texttt{end}_{\texttt{m4}}, \texttt{end}_{\texttt{m2}}, \texttt{m6}, \texttt{start}_{\texttt{m6}}, \texttt{m2}, \texttt{start}_{\texttt{m2}}, \texttt{m4}, \texttt{start}_{\texttt{m4}}, \texttt{end}_{\texttt{m4}}, \texttt{end}_{\texttt{m2}}, \texttt{end}_{\texttt{m6}}, \texttt{end}_{\texttt{m1}})$

Summary of Coverage Criteria The preceding discussion defines four different coverage criteria based on the IRCFG. Clearly, these criteria form a subsumption hierarchy. (Criterion c_i *subsumes* criterion c_j if complete coverage for c_i also achieves complete coverage for c_j.) The hierarchy is illustrated in Figure 2.

For the running example, the minimum number of complete IRCFG paths that achieve coverage for each criterion is as follows:

(All-IRCFG-Paths, 20)	(All-RCFG-Paths, 5)
(All-RCFG-Branches, 3)	(All-Unique-Branches, 2)

3.3 Handling of Cycles

The criteria were defined under the assumption that each RCFG is acyclic. If an RCFG contains a cycle, the number of RCFG paths is of course infinite. Due to space limitations, the handling of this case is discussed in detail elsewhere [19].

Since cycles are due to iterative behavior, the resolution of this issue depends on the chosen approach for loop testing. Many such approaches are possible [20, 3, 5]. In this work we consider one specific choice which is a simplified version of the traditional loop testing techniques described by Beizer [20]. In the future we plan to investigate other approaches for loop testing. Our current techniques can be easily generalized to support these approaches. Essentially, the only significant change will be related to adding information about loop bounds (i.e., minimum and maximum number of loop iterations).

Our current approach requires coverage of each RCFG subpath that represents (1) a possible iteration that does not lead to loop exit, and (2) a possible iteration that leads to loop exit, including complete bypassing of the loop. To formalize this approach, we augment the techniques presented earlier in the following manner. First, we assume that an RCFG contains artificial "loophead" nodes. Each such node represents the entry point of some loop. Any RCFG edge from the body of the loop to the loophead represents the end of a complete iteration of the loop; such edges are commonly referred to as *back edges* [21]. An example of a loophead node (labeled `lh`) is shown in Figure 3a.

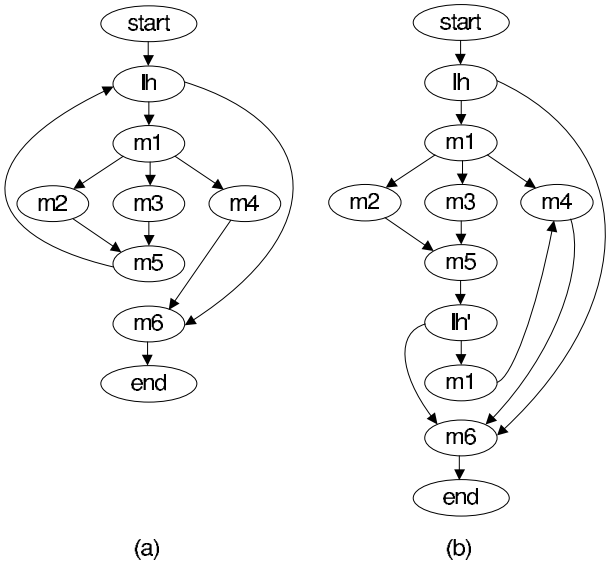

Fig. 3. Elimination of RCFG cycles.

Next, we define a graph transformation that takes an RCFG with cycles and produces an acyclic RCFG. The start-to-end paths in the transformed RCFG correspond to different execution scenarios for the corresponding cycle. The transformation is illustrated in Figure 3. We create a second artificial node $\texttt{lh}'$, and all back edges in the original cycle are redirected to the new node. As a result, the transformed graph does not contain any cycles. All exit edges from `lh` are also replicated: for example, we add an edge from $\texttt{lh}'$ to `m6`. Other nodes from the original cycle are replicated, if they lead to "premature" exits. For example, `m1` is replicated as a successor of $\texttt{lh}'$, and an edge is added from the replicated node to `m4`.

The paths in the transformed acyclic graph represent various execution scenarios for the original RCFG. For example, path $(\texttt{start}, \texttt{lh}, \texttt{m1}, \texttt{m2}, \texttt{m5}, \texttt{lh}', \texttt{m6}, \texttt{end})$ in the new RCFG represents the following execution in the original RCFG: the cycle is entered, it completes several full iterations (i.e., the back edge is traversed at least once), and then exits through (`lh`,`m6`). Furthermore, *at least one* of these full iterations follows the intra-cycle path (`m1`,`m2`,`m5`). It is easy to define when an RCFG paths in the original graph covers a path in the new graph. For example,

$$(\texttt{start}, \texttt{lh}, \texttt{m1}, \texttt{m2}, \texttt{m5}, \texttt{lh}, \texttt{m1}, \texttt{m3}, \texttt{m5}, \texttt{lh}, \texttt{m1}, \texttt{m4}, \texttt{m6}, \texttt{end})$$

in the original graph covers two paths in the new graph:

$$(\texttt{start}, \texttt{lh}, \texttt{m1}, \texttt{m2}, \texttt{m5}, \texttt{lh}', \texttt{m1}, \texttt{m4}, \texttt{m6}, \texttt{end})$$
$$(\texttt{start}, \texttt{lh}, \texttt{m1}, \texttt{m3}, \texttt{m5}, \texttt{lh}', \texttt{m1}, \texttt{m4}, \texttt{m6}, \texttt{end})$$

The graph transformation is easy to perform, by creating the replicated nodes and exit edges and by redirecting the back edges to `lh'`. In the presence of nested cycles, the innermost cycles can be processed first, then their enclosing cycles can be transformed, and so on. For brevity, we do not discuss the details of this straightforward algorithm.

The rationale behind the transformation is to make explicit the conceptually different scenarios in the execution of the original cycle, and to represent these scenarios as acyclic paths in the transformed graph. A tester will have to consider each of those scenarios when defining test cases, and therefore the number of paths in the new graph is an indication of the required testing effort. For the example in Figure 3, there are six such scenarios. Of course, an alternative way to estimate testing effort is to simply consider the number of RCFG paths in the original graph that can cover all paths in the transformed graph. For example, two paths from Figure 3a are sufficient to cover all paths in Figure 3b. However, such estimation does not take into account the complexity of the control flow inside the cycle. For example, consider an RCFG whose entire body is a cycle (except for the start and end nodes). Suppose the cycle has a single exit edge and complicated internal flow of control. *One single* multi-iteration path in the original RCFG is sufficient to cover all paths in the transformed graph. However, the effort to construct this path during testing depends on the internal structure of the cycle, and therefore can be better estimated by the number of paths in the transformed graph.

4 Run-Time Coverage Analysis

This section defines a coverage analysis for *All-RCFG-Paths*, *All-RCFG-Branches*, and *All-Unique-Branches*. We are in the process of building a coverage tool for these criteria, and this paper describes the design of the run-time analysis algorithm used in the tool. For brevity, the description outlines the ideas behind the algorithm without providing an in-depth discussion of all relevant details. At present we have no plans to implement coverage tracking for *All-IRCFG-Paths*, because the experimental results presented later in the paper raise questions about the practicality of this criterion.

The code instrumentation required to perform the run-time tracking is fairly straightforward. Immediately before each call site, we insert instrumentation to identify the method that is about to be invoked. We also insert instrumentation immediately after each call site, in order to know at run time that the invocation has just completed. The run-time events triggered by the instrumentation are used to traverse the IRCFG while the tests are being executed. The analysis maintains a "current" RCFG node which reflects the current state of the run-time execution. Immediately before a call site is about to make a call, the corresponding interprocedural edge in the IRCFG is traversed downwards and the current node is changed to the start node of the RCFG for the called method. The execution within the callee method proceeds until the flow of control reaches the exit of that method. At this point of time, the current node in the coverage

analysis is `end` in the RCFG for the callee. The return to the caller triggers an instrumentation event which shows that the call has just completed. As a result, the current node becomes the corresponding RCFG node in the caller method.

Based on the current RCFG node in the analysis, it is easy to compute coverage metrics for *All-RCFG-Branches* and *All-Unique-Branches*. To compute path coverage for *All-RCFG-Paths*, we use a variation of an approach for intraprocedural path profiling proposed by Ball and Larus [22]. Their technique assigns a unique integer path id to each distinct start-to-end path in a CFG. Instrumentation at CFG edges is used to update the value of a run-time integer accumulator. At CFG exit the accumulator contains the id of the executed path. We can use a similar technique for RCFG path tracking: each RCFG has an associated accumulator, which is initialized every time the flow of control enters the start node of the graph.

5 Minimum Number of Paths

In this section we define techniques for estimating the testing effort inherent in each of the four criteria discussed earlier. Given some IRCFG, for each criterion c we want to compute *a lower bound on the number of complete IRCFG paths* whose run-time coverage would guarantee the best possible coverage for c. This bound is an indication of how many complete IRCFG paths a tester may need to consider for coverage in order to satisfy c.

Complete IRCFG Paths First, what is the total number of complete IRCFG paths in a given IRCFG? The computation of the number of paths can be done in bottom-up fashion on the RCFG tree. Starting from the leaves, we can compute the number of IRCFG subpaths in each subtree. Consider some RCFG R in the tree, and suppose that we have already computed the number of IRCFG subpaths for each of the subtrees rooted at R's children. To compute the number of subpaths for the subtree rooted at R, we can traverse R in topological sort order. During the traversal, when we visit an RCFG node n in R, we compute the number $p(n)$ of all IRCFG subpaths from the start node of R to n. In the beginning of the traversal, $p(start_R) = 1$ for the start node of R. For each visited node n, we have

$$p(n) = \sum_{(n',n)\in R} p(n') \times q(n')$$

Here n' is an intraprocedural predecessor of n and $q(n') = \sum_{R'} p(end_{R'})$ where the sum is over all RCFG R' that are called by n' (i.e., there is an interprocedural edge from n' to the start node of R'). In the case when there are no such R', let $q(n') = 1$.

In this computation, for each intraprocedural edge (n', n) in R, we consider the number of IRCFG subpaths $p(n')$ from the start of R to n'. For each RCFG R' that is called by n', we examine the value $p(end_{R'})$ computed earlier for the end node of R'. There are a total of $p(n') \times p(end_{R'})$ IRCFG subpaths that start

at the beginning of R, lead to n', continue downwards into R', and eventually return back to n in R. The total number of complete IRCFG paths is the value $p(n)$ computed for the end node of the root RCFG.

Coverage of RCFG Branches To find the minimum number of complete IRCFG paths that contain all RCFG edges, we define an integer linear programming problem. Consider some hypothetical set S of complete IRCFG paths. For each RCFG edge e, let the integer value $v(e) \geq 0$ represent the number of times e is covered by all paths in S (i.e., the edge frequency of e in S). For each call node n in the IRCFG, we define the equation

$$\sum_{e \in In(n)} v(e) = \sum_{e \in Out(n)} v(e) \tag{1}$$

Here $In(n)$ denotes the set of all intraprocedural edges (n', n), and $Out(n)$ is the set of all intraprocedural edges (n, n''). Equation (1) shows that the number of times n is entered by paths in S is equal to the number of times n is exited.

For each call node n that has outgoing interprocedural edges, we also need to model the execution of the corresponding children RCFGs. This is done by

$$\sum_{e \in In(n)} v(e) = \sum_{e \in Call(n)} v(e) \tag{2}$$

Here $Call(n)$ denotes the set of all interprocedural edges $(n, start)$ entering the children RCFGs. Equation (2) encodes the fact that the number of times n is covered by S is equal to the number of times the children graphs are covered by S. We also model the execution frequencies of the edges coming out of each start node, using equation

$$v(e') = \sum_{e \in Out(start)} v(e) \tag{3}$$

where e' is the single interprocedural edge entering *start*.

For the *All-RCFG-Branches* criterion, we define a system that combines (1), (2), and (3) with the following equation:

$$v(e) \geq 1$$

for each edge e in each RCFG. Given this system, we solve a linear programming problem that minimizes the objective function

$$\sum_{e \in Out(start_{root})} v(e)$$

where $start_{root}$ is the start node of the root RCFG. This value represents the total number of times the start node is traversed by S, which is equal to the size of S. Let p^* be the minimum value for the objective function, as computed by a linear programming solver. It can be proven that p^* is the minimum number of complete IRCFG paths that contain all RCFG edges.

Unique Branches When considering unique RCFG edges, (1), (2), and (3) are combined with equation

$$v(e_1) + v(e_2) + \ldots + v(e_k) \geq 1$$

Here e_i are RCFG edges that are equivalent: they belong to different RCFGs for the same method, and all of them represent transitions between equivalent pairs of RCFG nodes. It can be proven that a linear programming problem with the same objective function as before produces the minimum number of complete IRCFG paths that contain each unique RCFG edge.

Coverage of RCFG Paths Recall that an RCFG path is a start-to-end sequence of intraprocedural edges inside an RCFG. Let S_R denote the set of all such paths in some RCFG R. For each edge $e \in R$, let $w(e)$ be the number of times e occurs in S_R. Suppose we combine (1), (2), and (3) with the following equation

$$v(e) \geq w(e)$$

for each edge e in each RCFG. Using the same objective function as before, it can be proven that a linear programming solver will produce the minimum number of complete IRCFG paths that cover all RCFG paths.

The proof of this property can be constructed by showing that (1) there exists a set of p^* complete IRCFG paths that covers each RCFG path, and (2) p^* is the smallest such number. Due to the rather technical nature of the proof, especially for property (1), we omit a detailed discussion of the proof technique. The intuition behind the proof is that even though the linear programming formulation enforces only edge-related properties (since it constraints only the values for edge frequencies), these properties are strong enough to guarantee path-related properties and ultimately RCFG path coverage.

To construct the system, we need to compute $w(e)$. Given an RCFG R, the values of $w(e)$ for all $e \in R$ can be computed in time linear in the size of R. First, a topological sort order traversal is used to compute the number $p'(n)$ of paths from the start node of R to any node $n \in R$. Clearly, $p'(n)$ is equal to the sum of $p'(m)$ for all predecessor edges $(m, n) \in R$. Similarly, using a traversal in reverse topological sort order, we can compute the number $p''(n)$ of paths from n to the end node of R. For an edge $e = (n_i, n_j)$, the value $w(e) = p'(n_i) \times p''(n_j)$.

6 Experimental Study

The approach described in this paper was implemented as part of the ongoing work on the RED tool for reverse engineering of sequence diagrams. The goal of this tool is to provide high-quality support for reverse engineering of UML sequence diagrams from Java code and for testing based on such diagrams. The tool uses several static analyses, including call graph construction [23, 17], call chain analysis [24], control flow analysis [13], and object naming analysis [14]. IRCFG construction was implemented as a straightforward extension of these

Table 1. Subject components.

Component	Methods	IRCFGs	Component	Methods	IRCFGs
`checked`	15	3	`pushback`	20	11
`bigdecimal`	33	26	`vector`	38	22
`gzip`	41	11	`boundaries`	74	13
`io`	86	12	`zip`	118	38
`decimal`	136	30	`date`	136	37
`calendar`	152	60	`collator`	157	17
`message`	176	59	`math`	241	156
`jflex`	313	93	`sql`	350	22
`mindbright`	488	161	`bytecode`	625	333

existing analyses. The lower bounds described in Section 5 were computed with the `lp_solve` linear programming solver (`groups.yahoo.com/group/lp_solve`).

The 18 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. Columns labeled "Methods" show the number of non-abstract methods in each component. For each component, we considered the set of methods that would normally be used to access the functionality provided by that component. For each such method we constructed an IRCFG starting at the method (i.e., the root RCFG was for this method). RED uses a parameter k to control the length of call chains in the reverse-engineered diagrams. Given some k, the number of messages in call chains is restricted to be at most k—that is, the depth of the corresponding RCFG tree is at most k, where the depth for the root is 0. We ran all experiments with the value $k = 3$. RCFGs were created only for component methods: if a component method called code external to the component, the corresponding RCFG node did not have a child RCFG. This restriction is part of the design of RED, and it allows a tool user to define a "scope of interest" and to ignore code that is outside of this scope. Columns "IRCFGs" show the number of IRCFGs that had non-trivial flow of control: at least one RCFG node had two or more outgoing edges. The total number of such IRCFGs for all components was 1104.

For each IRCFG counted in columns "IRCFGs" in Table 1, we determined the minimum number of complete IRCFG paths for the different criteria, as described in Section 5. Table 2 shows the distribution of these numbers for the entire set of 1104 IRCFGs. Each column shows the percentage of IRCFGs for which the minimum number of complete IRCFG paths was in the corresponding range. For example, the last column shows the percentage of IRCFGs that had a minimum number of complete paths greater than 1000.

The results from Table 2 lead to some interesting observations. In a substantial number of cases, the number of complete IRCFG paths is rather large. In fact, for several IRCFGs this number is very large (e.g., more than a million). Thus, even for the limited diagram depth of $k = 3$, and with the limited scope of the diagrams to component-only code, in many cases the *All-IRCFG-Paths*

Table 2. Minimum number of IRCFG paths.

Criterion	1–5	6–10	11–100	101–1000	>1000
IRCFG-Paths	29.1%	10.3%	16.8%	10.2%	33.6%
RCFG-Paths	40.8%	14.9%	27.4%	2.6%	14.2%
RCFG-Branches	45.5%	19.6%	31.9%	2.9%	0.2%
Unique-Branches	49.9%	22.1%	27.4%	0.5%	0.0%

Table 3. Reduction in the number of paths.

Ratio	1	$(1, 2]$	$(2, 10]$	$(10, 10^3]$	$> 10^3$
$\frac{IRCFG-Paths}{RCFG-Paths}$	35.1%	13.0%	12.7%	20.2%	19.0%
$\frac{RCFG-Paths}{RCFG-Branches}$	51.3%	26.2%	7.6%	13.2%	1.7%
$\frac{RCFG-Branches}{Unique-Branches}$	65.6%	23.5%	10.8%	0.2%	0.0%

criterion is clearly impossible to achieve in practice. These results confirm experimentally Binder's intuition [3] that the number of all start-to-end paths may be too large. The use of less demanding coverage criteria is one way to address this problem. Our results indicate that the three other criteria require less testing effort, and therefore are useful alternatives to *All-IRCFG-Paths*. For example, for *All-Unique-Branches*, almost all IRCFGs have a minimum number of paths that is ≤ 100, and for half of the IRCFGs this number is ≤ 5. The results suggest that each criterion provides a different tradeoff between testing effort and test comprehensiveness, and therefore a tester may benefit from having tool support for each criterion.

For each IRCFG counted in columns "IRCFGs" in Table 1, we computed the ratios between the minimum number of paths for different pairs of criteria, as shown in the first column of Table 3. Each of the remaining columns in that table shows the percentage of IRCFGs for which the ratio was in the corresponding range. For example, the last number of the first row in the table shows that for 19% out of the 1104 IRCFGs, the minimum number of complete IRCFG paths for *All-RCFG-Paths* is more than 1000 times smaller than the total number of complete IRCFG paths. The results in Table 3 are an indication of the reduction of testing effort when replacing a stronger criterion with a weaker one. All pairs of criteria exhibit substantial degrees of reduction, and the most significant change is from *All-IRCFG-Paths* to *All-RCFG-Paths*.

The results of the study can be summarized as follows. First, there is strong indication that the number of start-to-end paths in reverse-engineered sequence diagrams is often quite large, and therefore simpler (and easier to achieve) criteria should be available as options to testers. Second, the remaining three criteria appear to be good candidates for such options because they provide different tradeoffs for testing effort and comprehensiveness.

7 Related Work

As discussed in Section 2, several testing approaches are based on interaction diagrams that are constructed during analysis or design [3, 15, 4, 16, 6, 5, 7]. Our work applies similar techniques to diagrams that are constructed automatically from existing code. We define a spectrum of coverage criteria that could provide a tester with several options for the targeted test coverage.

The IRCFG used in our approach is based on two popular data structures: interprocedural CFG [25] and calling context tree [26]. An interprocedural CFG contains the CFGs for individual procedures, as well as edges connecting these CFGs. Unlike an IRCFG, an interprocedural CFG contains nodes for all statements in the procedures, and the edges between the individual CFGs do not form a tree. In a calling context tree, a node represents a procedure and the chain from the node to the tree root represents a call chain for that procedure. Similarly, the RCFGs in our approach form a tree that represents call chains.

Binder's all-branches approach [3] is based on a flow-graph representation of a sequence diagram which is similar to an RCFG. The discussion of the approach is limited to a single method, while our IRCFG combines information about several methods and their calling relationships. Briand and Labiche [5] represent an UML activity diagram with a directed graph in which paths correspond to sequences of use case that are considered for testing. The sequence diagram for a use case is represented by a regular expressions that captures the possible sequences of messages in the diagram. In order to automate the construction of the regular expression, the authors suggest modeling the sequence diagram with a labeled graph in which labels correspond to messages, similarly to our use of the RCFGs.

The traversal of the RCFG tree during the run-time analysis is similar to the dynamic profiling analyses from [26, 24]: in both cases, the sequence of methods on the run-time call stack is "simulated" by the analysis. The coverage of intra-RCFG paths uses the efficient techniques for path profiling from [22], with the appropriate modifications to ignore statements irrelevant to calls. Melski and Reps [27] present a general approach for interprocedural paths profiling which may be possible to adapt in order to obtain run-time coverage information for complete IRCFG paths.

8 Conclusions and Future Work

This work presents a family of control-flow-based coverage criteria for testing of object interactions in reverse-engineered sequence diagrams, together with a corresponding run-time coverage analysis. The experimental study highlights the inherent difficulty of criteria based on sequences of messages (i.e., path coverage). The study also indicates that less demanding criteria (e.g., based on branch coverage) may be a more practical choice for testing of object interactions. In our future work we plan to measure the coverage for these criteria that is achieved by real-world test suites, and to investigate the test weaknesses exposed by the different coverage statistics.

References

1. Binder, R.: Testing object-oriented software: a survey. Journal of Software Testing, Verification and Reliability **6** (1996) 125–252
2. Perry, D., Kaiser, G.: Adequate testing and object-oriented programming. Journal of Object-Oriented Programming **2** (1990) 13–19
3. Binder, R.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley (1999)
4. Abdurazik, A., Offutt, J.: Using UML collaboration diagrams for static checking and test generation. In: International Conference on the Unified Modeling Language. (2000) 383–395
5. Briand, L., Labiche, Y.: A UML-based approach to system testing. Journal of Software and Systems Modeling **1** (2002)
6. Fraikin, F., Leonhardt, T.: SeDiTeC—testing based on sequence diagrams. In: International Conference on Automated Software Engineering. (2002) 261–266
7. Wu, Y., Chen, M.H., Offutt, J.: UML-based integration testing for component-based software. In: International Conference on COTS-Based Software Systems. (2003)
8. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1999)
9. Fowler, M.: UML Distilled. 3rd edn. Addison-Wesley (2003)
10. Larman, C.: Applying UML and Patterns. 2nd edn. Prentice Hall (2002)
11. Kollman, R., Gogolla, M.: Capturing dynamic program behavior with UML collaboration diagrams. In: European Conference on Software Maintenance and Reengineering. (2001) 58–67
12. Tonella, P., Potrich, A.: Reverse engineering of the interaction diagrams from C++ code. In: IEEE International Conference on Software Maintenance. (2003) 159–168
13. Rountev, A., Volgin, O., Reddoch, M.: Control flow analysis for reverse engineering of sequence diagrams. Technical Report OSU-CISRC-3/04-TR12, Ohio State University (2004)
14. Rountev, A., Connell, B.H.: Object naming analysis for reverse-engineered sequence diagrams. In: International Conference on Software Engineering. (2005) to appear.
15. Jorgenson, P., Erickson, C.: Object-oriented integration testing. Communications of the ACM **37** (1994) 30–38
16. Basanieri, F., Bertolino, A.: A practical approach to UML-based derivation of integration tests. In: 4th International Quality Week Europe. (2000)
17. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology (2004) to appear.
18. Bodik, R., Gupta, R., Soffa, M.L.: Refining data flow information using infeasible paths. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering. (1997) 361–377
19. Rountev, A., Kagan, S., Sawin, J.: Coverage criteria for testing of object interactions in sequence diagrams. Technical Report OSU-CISRC-12/04-TR68, Ohio State University (2004)
20. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold (1990)
21. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)

22. Ball, T., Larus, J.: Efficient path profiling. In: IEEE/ACM International Symposium on Microarchitecture. (1996) 46–57
23. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for Java based on annotated constraints. In: Conference on Object-Oriented Programming Systems, Languages, and Applications. (2001) 43–55
24. Rountev, A., Kagan, S., Gibas, M.: Static and dynamic analysis of call chains in Java. In: ACM SIGSOFT International Symposium on Software Testing and Analysis. (2004) 1–11
25. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In Muchnick, S., Jones, N., eds.: Program Flow Analysis: Theory and Applications. Prentice Hall (1981) 189–234
26. Ammons, G., Ball, T., Larus, J.: Exploiting hardware performance counters with flow and context sensitive profiling. In: ACM SIGSOFT Conference on Programming Language Design and Implementation. (1997) 85–96
27. Melski, D., Reps, T.: Interprocedural path profiling. In: International Conference on Compiler Construction. LNCS 1575 (1999) 47–62