Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

Olga Volgin
University of Michigan
onv@eecs.umich.edu

Miriam Reddoch
Hewlett Packard
miriam.reddoch@hp.com

ABSTRACT

UML sequence diagrams are commonly used to represent the interactions among collaborating objects. Reverse-engineered sequence diagrams are constructed from existing code, and have a variety of uses in software development, maintenance, and testing. In static analysis for such reverse engineering, an open question is how to represent the intraprocedural flow of control from the code using the control-flow primitives of UML 2.0. We propose simple UML extensions that are necessary to capture general flow of control. The paper describes an algorithm for mapping a reducible exception-free intraprocedural control-flow graph to UML, using the proposed extensions. We also investigate the inherent trade-offs of different problem solutions, and discuss their implications for reverse-engineering tools. This work is a substantial step towards providing high-quality tool support for effective and efficient reverse engineering of UML sequence diagrams.

1. INTRODUCTION

The Unified Modeling Language (UML) has become a de facto standard for modeling of different aspects of software structure and behavior. *Sequence diagrams* are key UML artifacts for representing the behavior of a software system [9]. 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") [9]. An example of a sequence diagram is shown in Figure 1b.

1.1 Reverse-Engineered Sequence Diagrams

Various software tools provide support for reverse engineering, often based on UML design models. In UML tools, reverse engineering through static code analysis is typically restricted to *class diagrams*. The reverse engineering of *sequence diagrams* through static analysis is the next logical step for these tools. The need to generate sequence diagrams from program code is important enough that a request to UML tool vendors to provide such advanced functionality is included in one popular book on modern software development [6]. Some commercial modeling tools already incorporate reverse engineering of sequence diagrams (e.g., Together ControlCenter by Borland and EclipseUML by Omondo).

Reverse-engineered sequence diagrams can play a role in the maintenance of complex object-oriented systems. System evolution is often problematic in the absence of the original designers and developers due to incomplete or non-existent design information. Reverse-engineered diagrams provide essential insights for *software understanding* of such systems. Sequence diagrams are also the basis of several approaches for *testing of object-oriented software* [1]. These approaches test the interactions among collaborating objects, and sequence diagrams can be used to identify the interactions that must be covered. A coverage tool can employ static analysis to extract sequence diagrams from the tested code, and then use them to perform run-time coverage analysis during the execution of the given tests [13].

1.2 Using UML Control-Flow Primitives

Despite the significant practical importance of reverse-engineered sequence diagrams, there is a limited body of work on static analyses for constructing such diagrams. The work in this paper is part of the ongoing effort to build the RED tool for reverse engineering of sequence diagrams[1]. The goal of the tool is to provide comprehensive, effective, and efficient reverse engineering of UML sequence diagrams through static analysis of Java code.

The effort to build this tool revealed many challenging static analysis problems, and led to the definition and implementation of several analyses such as call chain analysis [12] and object naming analysis [11]. One of the key issues we had to address early in this project was the following:

> *How should the intraprocedural flow of control in the code be represented in the reverse-engineered sequence diagrams?*

Intraprocedural behavioral features (e.g., conditional and iterative behavior) presented some challenging problems for the tool. Attempts to use first-generation UML (versions 1.x) showed that the UML control-flow primitives do not provide enough expressive power to represent correctly the full generality of intraprocedural flow of control.

This experience motivated the work described in this paper. We use second-generation UML (version 2.0), which defines a richer set of control-flow primitives for sequence diagrams [9]. The following questions are considered:

- *Theoretically, is it possible to use UML 2.0 to represent general intraprocedural flow of control?* Even with the new features in version 2.0, the answer to this question is still negative.
- *What minimal extensions to UML 2.0 could be introduced to allow handling of general flow of control?* We

[1]Details are available at `presto.cse.ohio-state.edu/red`

identify two simple UML extensions such that it becomes possible to represent the flow of control in an arbitrary reducible exception-free control-flow graph (CFG).

- *How should a CFG be mapped to UML?* Using the proposed extensions, we define a novel control-flow analysis for mapping a control-flow graph to UML. To the best of our knowledge, this is the first time an analysis for this problem has been presented.

The analysis algorithm has two important properties. First, it is *general*: the approach handles any reducible exception-free CFG. Second, it is *precise*: the produced diagrams represent precisely the control-flow semantics of the code. This paper outlines the basic ideas of the algorithm; a detailed description is available elsewhere [14].

We consider this algorithm to be a first step in solving the problem of representing intraprocedural behavior in reverse-engineered sequence diagrams. Future work can easily leverage our approach. For example, if a tool designer decides that some control-flow details should be omitted for the sake of easier diagram comprehension, she can augment our technique with filtering mechanisms for these particular details, while ensuring that all of the remaining flow of control is still represented in the final diagram.

The work on the algorithm highlighted the following tradeoff for Red: in the general case, either the tool can use standard UML control-flow primitives, or it can represent precisely the flow of control in the code, but not both. Another important issue is that the same CFG node may have to be represented multiple times in the diagram. Thus, a tool can either represent precisely the flow of control, or it can create a diagram in which CFG nodes are not replicated, but not both. In addition to a discussion of these two tradeoffs, we present experimental results to determine how often the need for such tradeoffs occurs in practice. These results provide new insights for the creators of reverse-engineering tools for UML sequence diagrams.

1.3 Contributions

- **UML extensions:** We identify two simple and intuitive UML extensions that are sufficient to capture general flow of control.

- **Analysis algorithm:** The paper outlines an algorithm for precisely mapping intraprocedural flow of control to UML, using the proposed extensions. Experimental results indicate that the analysis has practical cost.

- **Investigation of tradeoffs:** We describe two inherent tradeoffs of the problem, and discuss their implications for reverse-engineering tools. Our experiments indicate that these tradeoffs could have significant impact for realistic Java software.

2. BACKGROUND

Given a set of Java classes, a Red user can choose a method `m` from these classes and can generate a sequence diagram that represents the interactions triggered by an invocation of `m`. For each method that is shown in the diagram, the control flow analysis examines the CFG of the method and creates a method-level data structure that encodes relevant aspects of the method's control-flow behavior. Subsequent display of the reverse-engineered diagram (implemented in a prototype visualization tool [15]) uses the data structures created for the individual methods. The tool interface allows a user to explore the diagram interactively—e.g., fragments can be collapsed and expanded on demand, and they can be filtered out based on user-defined criteria.

Note that the separate method-level data structures are simply building blocks for the entire multi-method diagram. This paper discusses *only* the problem of constructing the data structure for an individual method; related problems (e.g., inter-method control flow) are described in [11, 12]. The results of all static analyses (including the control-flow analysis) are combined in a single diagram which contains the appropriate messages, objects, object lifelines, control-flow information, etc.

By analyzing a method's CFG, we define a general approach that is independent of the peculiarities of any specific programming language. Thus, it would be trivial to use our algorithm in tools for languages other than Java, since CFG construction is simple to design and implement. Our approach could be used even in the absence of source code, as long as the object code can be analyzed to construct CFGs.

2.1 Running Example

Consider some method `m` in a class `X`, and suppose that the CFG of `m` is as shown in Figure 1a. The structure of this CFG is loosely based on methods from the standard library package `java.text`. For brevity, the CFGs for the methods called by `m` are not shown. The shaded nodes 5, 6, and 13 represent statements that are irrelevant to sequence diagrams (e.g., `i = 5;`). Typically, in real code most CFG nodes are irrelevant to the diagram.

Suppose that the tool user wants a sequence diagram that represents the interactions triggered by a call to `m`. Figure 1b shows an example of such a diagram. For the sake of the example, assume that the calls to `p1`, . . . , `p5` are made by methods `m2` through `m7`, and no other calls exist in any of the CFGs for any of the methods. The *loop*, *opt*, *break*, and *alt* elements represent the flow of control during the interactions, as described shortly. These elements are based on the data structure produced by our CFG analysis.

2.2 UML Control-Flow Primitives

A sequence diagram contains objects and messages exchanged among them. UML 2.0 also defines *interaction fragments* as diagram entities that represent various aspects of the interaction [9]. 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 1.

An opt/loop/break fragment encloses an ordered sequence of other fragments. A sequence of fragments represents one or more sequences (traces) of run-time events [9]. 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 1 is guarded by the condition `!c2` corresponding to CFG edge $(4, 7)$.

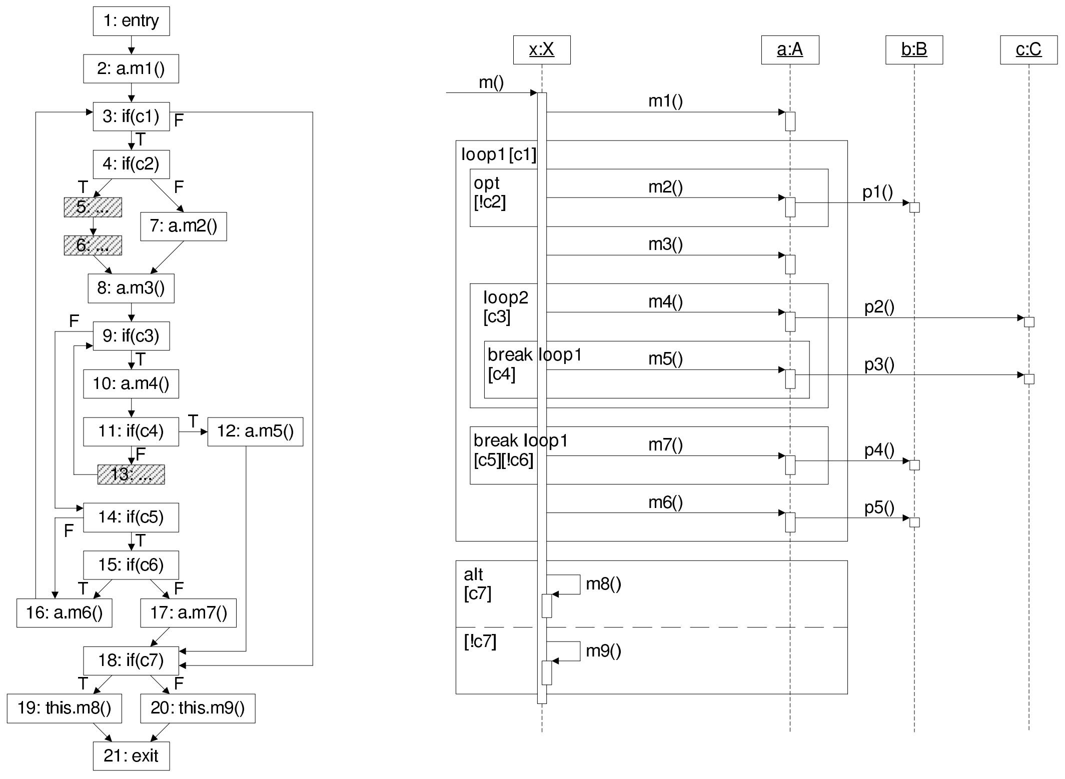

Figure 1: (a) Control-flow graph for a method `m` (b) Reverse-engineered sequence diagram for `m`

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 some 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 1 has two alternatives, the first one guarded by `c7` and the second one by `!c7`. An alternative inside an alt fragment could be empty: for example, when the corresponding behavior does not result in any messages being sent. If only one alternative is non-empty, the alt fragment is equivalent to an opt fragment.

The sequence enclosed in a loop fragment is repeated until the guard condition becomes false. For the outer loop in Figure 1, the sequence of fragments enclosed in the loop is repeated until `c1` becomes false. The loop can also exit through two break fragments. A break fragment represents a "breaking" scenario: first the fragment sequence inside the break fragment is executed, and then the execution of the fragment enclosing the break completes immediately.

2.3 Generalized Break Fragments

A break fragment, as defined by UML 2.0, breaks out of the immediately surrounding fragment. This definition makes it impossible to express the semantics of real-world code, in which control can "jump" over several levels of nesting. We propose a generalized break fragment that allows breaking out of multiple enclosing fragments. The fragment 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; Figure 1 illustrates this approach.

2.4 Multiple CFG Exits

A CFG could have multiple exit nodes, where each exit node is guarded by a condition. For example, "`if (c) { a.m(); return; }`" should be mapped to a fragment similar to a break fragment, with a guarding condition `c` and with the message `m` inside it. However, in this case the flow of control breaks out of the entire method and returns back to the caller. UML 2.0 does not define explicit notation for this situation. We define a *return fragment* which is similar to a break fragment. All "premature" CFG exits are represented by return fragments.

Strictly speaking, in some cases this effect could be achieved without return fragments; instead, artificial alt fragments could be used. For the example from above, it may be possible to have an alt fragment in which one alternative corresponds to `a.m()` and the other alternative corresponds to the remaining part of the CFG. In our experience, this approach produces deeply nested fragments that are very unnatural and hard to comprehend. Thus, we believe that the use of return fragments is essential for multi-exit CFGs.

2.5 Precision vs. Interoperability

A reverse-engineering tool can either use standard UML control-flow primitives, or it can represent precisely the flow of control in the code, but not both. Furthermore, the extent of non-standard UML additions determines the level of imprecision in the mapping to UML. Our decision was to use the UML extensions described earlier in order to obtain precise mappings from CFGs, since these extensions were conceptually simple and easy to visualize. However, if the results of RED need to be exported to other UML tools in the future (e.g., using tool-independent formats such as XMI), we will need to introduce precision-losing mappings back to standard UML 2.0. The creators of any similar reverse-engineering tool will also be faced with this issue, and will have to consider this precision-vs-interoperability tradeoff.

3. CONTROL-FLOW ANALYSIS

The two simple UML extensions described above are both necessary and sufficient to capture the full complexity of an arbitrary reducible exception-free CFG. Given the CFG, our analysis produces a set of interaction fragments that precisely represents the control-flow behavior which affects the messages being sent by this method. These fragments encode *all and only* sequences of call statements that occur along all CFG control-flow paths. This section provides a high-level overview of the analysis; more details are available in [14].

The analysis computes *branch successors* and *loop successors* for certain CFG nodes. This computation is based on the well-known notion of post-dominance. CFG node n_2 *post-dominates* n_1 if every path from n_1 to an 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 tree in which each parent node is the immediate post-dominator of its children.

3.1 Branches and 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. In this case it is necessary to determine which CFG nodes should be considered when building the contents of this new fragment. Intuitively, we need to determine where the fragment "stops"; this stopping point is the start of the fragment that will follow the new alt fragment. For example, for node 4 in Figure 1, the analysis will create an alt fragment.[2] The fragment following the alt in the loop's fragment sequence will be constructed starting from node 8, which is the "merge point" of the branches coming out of 4.

Consider a branch node n with outgoing edges (n, n_i). The branch successor of n is defined to be the lowest common ancestor of all n_i in the post-dominance tree for the CFG. A common ancestor represents a merge point for all branches coming out of n. The lowest such ancestor is the merge point that is the "closest" to n. The paths from n to this node define the CFG nodes that should be considered when building the contents of the alt fragment created for n.

If n is inside a loop, the notion of a branch successor must be restricted to the flow of control that stays within the loop. For this, we define the notion of *post-dominance inside a loop.* Consider nodes n_1 and n_2 in some loop L. Node n_2 post-dominates n_1 inside L if n_2 belongs to every loop-only path from n_1 to the loop header. This means that if n_1 is reached during some iteration of L and subsequently the iteration completes successfully—i.e., the header of L is eventually reached—then n_2 is reached after n_1 as part of that same iteration. If a node n has more than two successors n_i in its enclosing loop L, the branch successor of n is the lowest common ancestor of all such n_i in the post-dominance tree for L. For example, for node 14 in Figure 1, the branch successor is node 16. This definition can be easily generalized for nested loops.

[2]As shown in Figure 1, this fragment can later be transformed into an opt fragment, because one of its alternatives does not contain.

3.2 Loops and Loop Successors

The notion of a *reducible* CFG is standard in program analysis research. A loop in a reducible CFG is a strongly-connected subgraph L such that exactly one node $n \in L$ has a predecessor that is not in the loop. Node n is the header node of L. The control-flow features of Java ensure that the CFG of a method is reducible. UML 2.0 cannot represent precisely the semantics of code with irreducible CFGs, because a loop fragment has only one entry point.

Whenever the analysis encounters the header node of a loop, it creates a loop fragment. In this case, the analysis needs to decide which nodes should be considered when building the contents of this loop fragment. For example, for node 3 in Figure 1, the analysis will create a loop fragment. Node 18 is the merge point for the three different possible ways to exit the loop, and thus the analysis needs to consider all nodes occurring on paths from 3 to 18.

For each loop L we determine a *loop successor.* In the case of an outermost loop, the loop successor is the lowest common ancestor for all targets of loop exit edges in the method-level post-dominance tree. For example, for the outer loop in Figure 1, these targets are nodes 12, 17, and 18, and the loop successor is 18. This is the earliest common point for all possible executions after the loop terminates. The generalization for nested loops is presented in [14].

3.3 Multiple CFG Exits

The algorithm becomes more complicated in the presence of multiple CFG exits. In particular, it becomes necessary to compute and use information about control dependencies [3] between branch nodes and exit nodes, and to create return fragments for CFG subgraphs which lead to exit nodes. The general treatment of multiple exit nodes is described in [19]; our implementation fully handles this general case.

3.4 Exceptions

The handling of exceptional flow of control depends on the language mechanisms for creating such flow. In Java, exceptions may be *synchronous* (occurring at well defined program points) or *asynchronous* (occurring non-deterministically). Synchronous exceptions can be raised explicitly with a `throw` statement, or implicitly as the result of an expression evaluation, a linking/loading error, or a resource error. Asynchronous exceptions occur as the result of an error in the virtual machine. Synchronous exceptions may be *checked* or *unchecked.* Unchecked exceptions are instances of class `java.lang.RuntimeException`, `java.lang.Error`, or any of their subclasses; a typical example is `NullPointerException`. All other exceptions are considered to be checked exceptions.

Our algorithm essentially ignores exceptional flow of control in Java. First, implicitly-thrown exceptions (i.e., without explicit `throw`) are not considered. Since such exceptions could occur implicitly at a large number of program points, their representation in a diagram will produce significant visual clutter without providing any useful information. Thus, we believe that only explicitly-thrown exceptions should be considered for reverse-engi-neered diagrams. Second, we do not try to identify pairs of CFG nodes (n_1, n_2) such that n_1

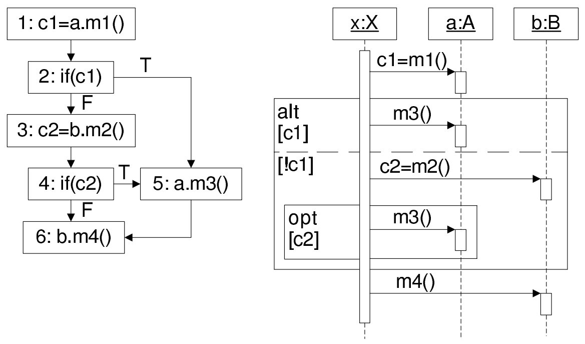

Figure 2: Message m3 is replicated.

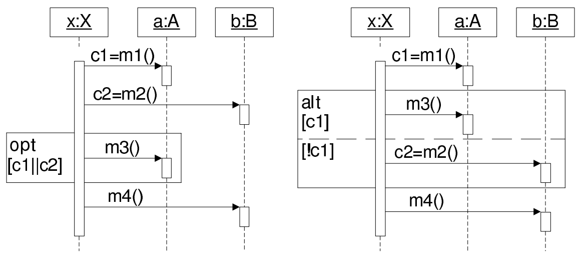

Figure 3: Possible imprecise mappings.

explicitly throws an exception using `throw`, and n_2 is a `catch` clause that may catch that exception. In most cases, n_1 and n_2 will belong to two different methods, and exception-flow analysis must be an interprocedural analysis (e.g., [16]). At present we do not employ such an analysis; as a result, our implementation ignores all `catch` clauses. Furthermore, it is unclear what UML notation should be used to represent the exceptional flow of control from n_1 to n_2.

In the case of a `throw` statement, the algorithm treats this statement as a CFG exit node and applies the same techniques we use for `return` statements, as described in Section 3.3. The resulting fragment is similar to a return fragment, but instead of leading to a method exit it should lead to a catch clause. In our future work we will investigate how this `throw` fragment can be associated with the corresponding `catch`, and how this association can be visualized in the diagram.

3.5 Fragment Construction

After computing branch/loop successors, the analysis traverses the CFG and creates the corresponding fragments. As nodes are encountered during the traversal, the current fragment sequence is "populated". New interaction fragments are created as necessary, and branch/loop successors are used to decide when to exit the current fragment. This approach is guaranteed to produce fragments that encode precisely all and only sequences of calls in the CFG [14].

4. MAPPING WITHOUT REPLICATION

Consider the following code fragment:

```
if (a.m1() || b.m2()) { a.m3(); }    b.m4();
```

Figure 2 shows the corresponding CFG subgraph and the interaction fragments produced by the analysis. Clearly, in this mapping from the CFG to UML, CFG node 5 is represented in the diagram two times, once in each of the alternatives of the alt fragment. We will say that node 5 is *replicated* by the mapping. It is easy to show that for this CFG, there does not exist a no-replication mapping which represents precisely the sequence of run-time events encoded by the CFG. There are several possible no-replication mappings that are not precise: two examples are given in Figure 3.

In the general case there will be CFGs for which a no-replication mapping does not exist. The next section presents experimental evidence that such CFGs do occur in real-world Java code. Intuitively, in these cases the "sequentiality" of the control-flow primitives makes the diagrams less expressive than the arbitrary flow of control in a CFG, and full generality can be achieved only through replication of CFG nodes.

Depending on the intended uses of reverse-engineered diagrams, tool designers have to decide on a particular tradeoff between size and precision. For example, for program understanding purposes it may be desirable to avoid replication altogether, even if this means that the diagrams represent only a subset of the possible run-time event sequences. ControlCenter and EclipseUML seem to have taken this approach. On the other hand, if the diagrams are used as the basis for test coverage measurements, precision may be more important than diagram size: e.g., if testing attempts to cover all possible sequences of messages [1, 13].

The full-precision choice and the no-replication choice define the two ends of the design spectrum. Tool builders could decide that a particular point somewhere in the middle of this spectrum is the appropriate choice. Tools could even implement multiple choices, and allow the user to adjust the tradeoff. This is the approach we plan to take in RED. The current implementation of RED allows mappings that have replication, in order to achieve full precision. We are investigating a set of precision-losing transformations that would allow systematic exploration of the spectrum of possibilities for this tradeoff. Furthermore, we will augment this work with effective visualization techniques that allow a tool user to investigate the diagrams. Our prototype provides highly-interactive diagram visualization and exploration [15], and it will be easy to incorporate user-defined tradeoff adjustments.

Another possibility is to introduce additional UML extensions (i.e., new control-flow primitives) for increased expressive power. A disadvantage of this approach is the "drift" from the standard, which will create problems for the interoperability with other UML tools. This is another decision that tool designers have to face: if interoperability is not a concern, it may be justifiable to add tool-specific UML extensions. We do plan to integrate our tool with other UML tools (including commercial ones), and we decided against using additional UML enhancements beyond the minimal extensions described in Section 2.

5. EMPIRICAL STUDY

This section summarizes some of the experimental results from our evaluation of the analysis; additional results are available in [14, 19]. The 20 subject components used in the study are listed in Table 1. The second column shows the number of non-abstract methods in each component.

First we considered the cost of the analysis. The third column in Table 1 shows the analysis running times, in seconds.

Component	Meth	Time (s)	(a)	(b)	(c)
`collator`	157	4.84	44.9%	4.3%	5.8%
`date`	136	5.43	54.2%	16.7%	41.7%
`decimal`	136	0.77	60.0%	16.0%	32.0%
`message`	176	1.33	45.0%	12.5%	32.5%
`boundaries`	74	0.54	71.4%	0%	0%
`gzip`	41	0.21	23.1%	0%	0%
`zip`	118	0.54	42.2%	3.0%	0%
`math`	241	0.96	89.9%	1.7%	8.4%
`pdf`	344	0.74	40.3%	0%	2.7%
`mindbright`	488	2.08	36.4%	4.0%	4.0%
`sql`	350	0.53	31.8%	9.1%	13.6%
`html`	298	1.42	51.8%	1.8%	6.3%
`jess`	627	2.83	39.2%	3.2%	11.1%
`io`	86	0.34	68.2%	0%	4.5%
`jflex`	313	14.65	59.5%	1.4%	2.7%
`bytecode`	625	6.65	41.4%	0%	7.6%
`checked`	15	0.12	33.3%	0%	0%
`bigdecimal`	33	0.25	100%	0%	0%
`vector`	38	0.19	53.3%	0%	0%
`pushback`	20	0.12	80%	0%	0%

Table 1: Subject components.

This is the total time to run the analysis for all methods in a component, on a 900 MHz Sun Fire 280-R machine. The results strongly suggest that the analysis cost is practical.

For each method m with a non-trivial body, we ran the control-flow analysis and used its output to answer the following questions:

- Is it necessary to use return fragments in order to obtain a precise mapping for m?
- Is it necessary to use a generalized break fragment (crossing multiple levels of fragment nesting) to obtain a precise mapping for m?
- Is it *im*possible to have a precise no-replication mapping for m, using only UML 2.0 and the extensions from Section 2?

Column (a) in Table 1 shows the percentage of methods for which a return fragment was a necessity in order to obtain a precise mapping. The results indicate that the number of such methods is substantial, and therefore the handling of multiple method exits is an important issue for reverse-engineered sequence diagrams. If return fragments (or some equivalent notation) are not used, the loss of information is likely to be substantial.

Column (b) shows the percentage of methods for which a precise mapping was possible only if we used a generalized break fragment that "jumps" out of multiple enclosing fragments. The results suggest that the handling of this situation is not as important as for return fragments. Nevertheless, CFGs that require such handling do occur in the subject components. In this situation a tool designer has two obvious choices. One possibility is to introduce the generalized break fragments described in Section 2. The second option is to use only "regular" UML 2.0 break fragments, and to ignore the flow of control that requires the generalized version. The control-flow analysis can be easily modified to support this second option: intuitively, whenever a CFG edge "jumps too far out", the edge can be simply ignored.

Column (c) contains the percentage of methods for which a precise no-replication mapping was not possible. For some components, this tradeoff is clearly not an issue. However, the overall conclusion from these experiments is that this tradeoff occurs frequently enough to justify careful future studies. To gain further insights, we examined the three components with the highest percentages. These components are from the standard Java package `java.text` and they contain highly-complex code for parsing and formatting of data. For code with complicated internal logic, we believe that the practical solution is to sacrifice some precision for the sake of reduced diagram complexity. As discussed in Section 4, we plan to define precision-losing transformations and to make them available for interactive user-defined tradeoff adjustments.

6. RELATED WORK

Several techniques employ dynamic analysis of run-time program behavior to perform reverse engineering of sequence diagrams or similar representations [17, 10, 4, 8, 2]. An advantage of these approaches is that they create diagrams that represent the actual behavior of the software. However, in many cases input data for run-time execution is not available, especially for incomplete systems (e.g., reusable modules) that cannot be executed in stand-alone manner. Furthermore, it is not known how well the execution covers all possible interactions. 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. As another example, sequence diagrams produced only with dynamic analysis cannot be used for evaluating the adequacy of testing. Some dynamic reverse-engineering analyses take into account conditions and iterations [17, 2], but the quality of the results depends on pattern matching heuristics that identify certain sequences of run-time events and attempt to extract from them control-flow primitives.

Reverse engineering of sequence diagrams through static analysis avoids these problems. The ControlCenter and EclipseUML modeling tools include such functionality as an advanced feature for support of round-trip engineering. These tools do not appear to handle correctly more complicated flow of control (e.g., due to break statements). Kollman and Gogolla [5] define a static analysis for reverse engineering of collaboration diagrams (similar to sequence diagrams), but do not discuss the representation of conditional and iterative behavior. Tonella and Potrich [18] present a static analysis for reverse engineering of sequence diagrams and collaboration diagrams from C++ code, but do not perform analysis of intraprocedural flow of control.

The Dava decompiler [7] uses a static analysis that maps CFGs to the control-flow constructs of Java. Our work has a similar goal, but the target are UML control-flow primitives, which are simpler and less expressive than those in Java. There are some significant technical differences between the two approaches. For example, in [7] the body of a conditional statement is determined by considering the nodes *dominated* by a branch node, while our approach considers *post-dominance* relationships to determine a branch successor and the scope of an alt fragment. Our algorithm traverses the hierarchical structure of the CFG in order. The technique in Dava uses a sequence of stages where each stage attempts to identify particular constructs (e.g., only loops).

7. CONCLUSIONS AND FUTURE WORK

This work describes a novel algorithm for mapping reducible exception-free control-flow graphs to UML interaction fragments. As part of RED, the analysis solves one important problem for reverse engineering of sequence diagrams. We plan to perform additional investigations of the tradeoffs discussed earlier, in order to find the right balance between precision and practicality for different uses of the analysis results—for example, for program understanding, for software testing, etc.

Acknowledgments. We would like to thank the PASTE reviewers for their helpful comments and suggestions.

8. REFERENCES

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

[2] 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.

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

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

[5] 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.

[6] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2nd edition, 2002.

[7] J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *Working Conference on Reverse Engineering*, pages 368–374, 2001.

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

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

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

[11] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *International Conference on Software Engineering*, pages 254–263, 2005.

[12] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, July 2004.

[13] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *Fundamental Approaches to Software Engineering*, LNCS 3442, pages 282–297, 2005.

[14] A. Rountev, O. Volgin, and M. Reddoch. Control flow analysis for reverse engineering of sequence diagrams. Technical Report OSU-CISRC-3/04-TR12, Ohio State University, Mar. 2004.

[15] R. Sharp and A. Rountev. Interactive exploration of UML sequence diagrams. In *IEEE Workshop on Visualizing Software for Understanding and Analysis*, 2005.

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

[17] 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.

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

[19] O. Volgin. Control flow analysis for reverse engineering of sequence diagrams. Master's thesis, Ohio State University, June 2004.