StaticGen: Static Generation of UML Sequence Diagrams

Chris Alvin[1(✉)], Brian Peterson[2], and Supratik Mukhopadhyay[2]

[1] Bradley University, Peoria, IL 61625, USA
calvin@bradley.edu
[2] Louisiana State University Baton Rouge, Baton Rouge, LA 70803, USA
{brian,supratik}@csc.lsu.edu

Abstract. UML sequence diagrams are visual representations of object interactions in a system and can provide valuable information for program comprehension, debugging, maintenance, and software archeology. Sequence diagrams generated from legacy code are independent of existing documentation that may have eroded. We present a framework for static generation of UML sequence diagrams from object-oriented source code. The framework provides a query refinement system to guide the user to interesting interactions in the source code. Our technique involves constructing a hypergraph representation of the source code, traversing the hypergraph with respect to a user-defined query, and generating the corresponding set of sequence diagrams. We implemented our framework as a tool, *StaticGen*, analyzing a corpus of 30 Android applications. We provide experimental results demonstrating the efficacy of our technique.

1 Introduction

Legacy object-oriented code may be accompanied by high-level documentation and/or descriptive source code comments, each of which may contain omissions or erroneous information. As documentation erodes, an engineer can trust only the source code. A necessary component of software archeology in object-oriented systems is the interactions among objects. A sequence diagram is a visual representation of those object interactions as well as their lifelines.

Sequence diagrams generated from legacy code are independent of existing documentation. Dynamic techniques for generation of sequence diagrams from legacy code [5,16,17,22,25] can synthesize a subset of all possible sequence diagrams based on runtime traces. Existing static techniques [34] result in sequence diagrams that replicate the original legacy source code, including conditionals and loops, without providing further intuitive notions beyond the code itself.

We present a technique, depicted in Fig. 1, for static generation of UML sequence diagrams together with a query system to guide the user to the most interesting interactions in the (unobfuscated) source code. Given an existing object-oriented code base as input, our technique involves three distinct steps as shown in Fig. 1. The first step in our technique (Fig. 1) takes the input code base and transforms it into a *typed control flow graph* (TCFG): a control flow

© Springer-Verlag GmbH Germany 2017
M. Huisman and J. Rubin (Eds.): FASE 2017, LNCS 10202, pp. 173–190, 2017.
DOI: 10.1007/978-3-662-54494-5_10

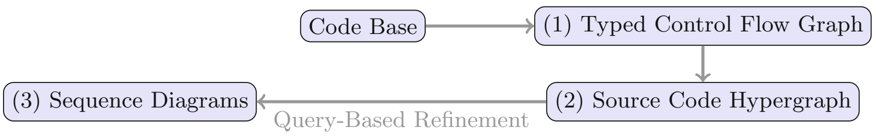

Fig. 1. The *StaticGen* system flowchart

```
public class Main extends ActionBarActivity {
   private int goodId, btnID = 2131296336; private Button b;
   private Random r = new Random();

   public boolean onOptionsItemSelected(MenuItem item) {
      int id = item.getItemId();
      return super.onOptionsItemSelected(item);
   }
   public void middleButtonOnClick(View v) {
      ((Button)v).setText("Clicked");

      int c = 0;
      if (r.nextBoolean()) c = getRed();
      else c = getBlue();

      int opt = r.nextInt(4);
      if (opt == 0) SetUpperLeftButton(c);
      else if (opt == 1) SetUpperRightButton(c);
      else if (opt == 2) SetLowerLeftButton(c);
      else SetLowerRightButton(c);
   }
   // Other Set methods omitted for redundancy
   private void SetUpperRightButton(int c) {
      b = (Button)findViewById(btnID);
      SetBtnColor(b, c);
   }
   private void SetBtnColor(Button b, int c) {
      b.setBackgroundColor(c);
      goodId = b.getId();
   }
   private int getRed() { return Color.RED; }
   private int getBlue() { return Color.BLUE; }
}
```

Fig. 2. Example android source code

graph annotated with type information—a familiar structure in static analysis acquired from an existing front-end tool such as Soot [29] or goto-cc [15].

The TCFG for a program P captures the execution of P, but does not capture (a) the interactions among the objects constituting P, (b) their context, and (c) the *causal ordering* of their interactions. Hence, the second step of our methodology involves constructing a directed *code hypergraph* [6, Ch. 1] (Sect. 3) that captures (1) intra- and inter-procedural control flow, (2) message *interactions* among objects, (3) message *context*, and (4) *causal ordering* of messages. From the source code in Fig. 2, we consider a portion of the generated *code hypergraph* (corresponding to a hyperpath [6, Ch. 1]) in Fig. 3. A code hypergraph corresponding to the input source code contains two categories of nodes. The first category refers to *code objects*: objects and their datatypes (rounded corners in Fig. 3). The second category of nodes, called *trace* nodes, capture a trace of a

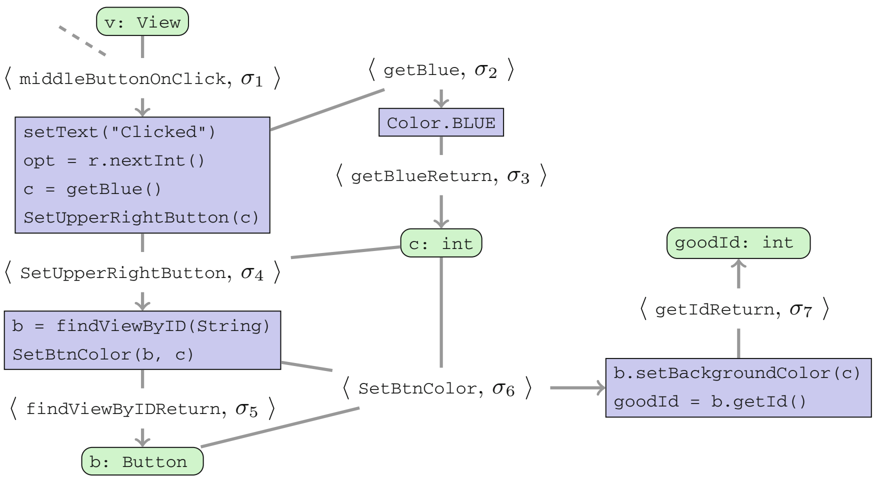

Fig. 3. A portion of the code hypergraph corresponding to the code in Fig. 2

method (rectangles in Fig. 3). For example, it is clear that `middleButtonOnClick` in Fig. 2 has $2 * 4 = 8$ possible traces due to the permutation of respective branches; Fig. 3 depicts one of those 8 trace nodes.

A directed hyperedge captures a message context in the form of an origin hypernode (a set of nodes) and causal ordering by virtue of directedness of hyperedges. The annotation of each hyperedge defines corresponding messages. For example, a call to `middleButtonOnClick` in Fig. 2 requires the context of an object of type `View` and a callee; the corresponding hyperedge in Fig. 3 is labeled accordingly with the destination method and program state information for context.

The third step in our technique (Fig. 1) constructs sequence diagrams (Sect. 4) given a *code hypergraph* corresponding to an input code base. Each hyperpath [6, Ch. 1] in that hypergraph encodes all object interactions in an execution of the code base and therefore a corresponding sequence diagram can be generated. The hyperpath in Fig. 3 corresponds to the sequence diagram shown in Fig. 4. To empower the user to identify 'interesting' interactions, we provide a *query-based refinement* interface that allows the user to narrow the resultant set of generated sequence diagrams based on their criteria and guides the user to the most interesting interactions in the source code.

We evaluated the effectiveness of our tool, *StaticGen*, on 30 open source Android applications [1,35]. *StaticGen* generated 647.1 sequence diagrams on average per package taking a mean of 96.78 s for each package. In addition to helping developers comprehend legacy code, *StaticGen* could fill an important security role for normal users as well. In a second experiment, we conducted a case study of using *StaticGen* to uncover *security vulnerabilities*. The query refinement system of *StaticGen* using the notions of 'interesting' and 'refinement', allowed us to narrow down the set of all sequence diagrams of a program to a subset that exposed a vulnerability.

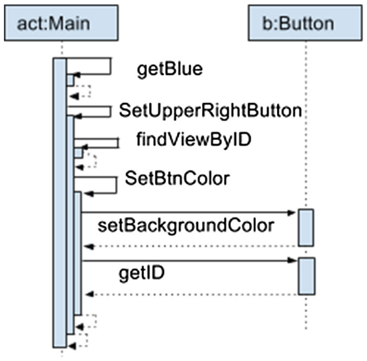

Fig. 4. Sequence diagram for an execution path in Fig. 2

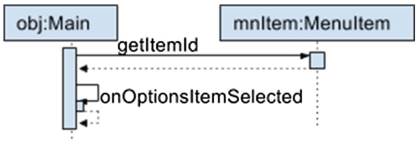

Fig. 5. Uninteresting sequence diagram for Fig. 2

This paper makes the following contributions:

- Section 2 formalizes a sequence diagram with respect to a hyperpath in a hypergraph.
- We describe a tool, *StaticGen*, for statically generating sequence diagrams by constructing (Sect. 3) and exploring (Sect. 4) a *code hypergraph* for an input code base.
- *StaticGen* provides a query system to refine the set of generated diagrams and guide the user to the most interesting interactions in the source code. (Sect. 5).
- We illustrate the efficacy of our technique (Sect. 6) with quantitative analyses and a case study to identify a security vulnerability.

2 Program Abstraction and Code Hypergraphs

In this section, we describe an abstract model for programs, formalize the notion of a *code hypergraph*, define a sequence diagram in that context, and define terms related to the features and quality of a given sequence diagram.

2.1 Program Abstraction Model

To define a framework for static generation of sequence diagrams not tied to a particular object-oriented language, we introduce a *typed control flow graph* (TCFG), an abstract model that will serve as the basis for our analysis. The model maintains both data flow (i.e. program points with state information attributed to collecting semantics, alias analysis, etc.) and control flow information (i.e. intra-procedural instructions and inter-procedural method calls). A program is abstracted by a typed control flow graph (Definition 1) containing two types of edges: intra-procedural transfer edges and inter-procedural call edges.

Definition 1 (Typed Control-Flow Graph). A typed control-flow graph for a program P is a control flow graph $G_{N_T} = (N_T, X, C, n_0)$ where N_T is the set of program points including *type* information for all variables, X is the set of intra-procedural transfer edges, C is the set of inter-procedural call edges, and n_0 is the entry point of the program.

For acyclic TCFGs, we assume the standard notion of *sequential ordering of instructions* as induced by the directed nature of the representative graph. We describe our approach in the context of a simple object-oriented programming language with conditionals, assignments, loops, references, and methods with call-by-value. We omit the details of the language as the operational and denotational semantics are defined in the usual way.

2.2 The Code Hypergraph

For a program P, we use a *directed hypergraph* [6, Ch. 1] data structure where *hypernodes* (sets of nodes) capture the context of interactions and *directed hyperedges* capture the interactions of objects constituting P. The order of hyperedges in a hypergraph captures the notion of *causal ordering* [2]; for events (invocations or returns of methods) U and V, we write $U \rightarrow V$ if event U is causally ordered before event V. In our model, hyperedges consist of a set of origin nodes and a single target node; a many-to-one relationship.

We formally introduce an abstract, many-to-one directed hypergraph called an *annotated hypergraph* where all hyperedges are annotated according to the problem space. A simple annotation may consist of a boolean expression indicating if a hyperedge is to be considered (in)active; that is, all the context information corresponding to the hyperedge is available or not.

Definition 2 (Annotated Hypergraph). An annotated hypergraph is a directed hypergraph $H\,(N, E_{\mathcal{A}})$ where N is a set of nodes and $E_{\mathcal{A}} \subseteq 2^N \times N \times \mathcal{A}$ a set of directed annotated hyperedges over a set of annotations $\mathcal{A}$. Each directed hyperedge $e \in E_{\mathcal{A}}$ is defined as an ordered pair $e = (S, t, A)$ where $S \subseteq N$, $t \in N$, and $A \in \mathcal{A}$.

Hyperedge annotations correspond to events in the program. Given two hyperedges, $E_A = (S_1, t_1, A)$ and $E_B = (S_2, t_2, B)$ with origin hypernodes S_1 and S_2, respectively, t_1 and t_2 target nodes, respectively, and annotations A and B respectively, we say $A \twoheadrightarrow B$ if $t_1 \in S_2$. We define $\rightarrow$ to be the transitive closure of $\twoheadrightarrow$. An important component of our technique is the hyperpath construction; we define hyperpath in the context of an annotated hypergraph.

Definition 3 (Hyperpath). Let $H\,(N, E_{\mathcal{A}})$ be an annotated hypergraph, $G \subset N$, and $g \in N$. A *hyperpath* Y (of length n) from G to g is a sequence of hypernodes $G_0, G_1, G_2, \ldots, G_{n-1}$ where $G_0 = G$ and $G_{n-1} = \{g\}$ such that for each $1 \leq i \leq n-1$ there exists a hyperedge $(G_{i-1}, g_i, A_i) \in E$ where $g_i \in G_i$ and $A_i \in \mathcal{A}$.

The annotated hypergraph in Definition 2 is an abstract structure that we instantiate to encode interactions, context, and causal ordering through nodes and hyperedges. We call the resulting hypergraph a *code hypergraph*. Before we formally define a *code hypergraph*, we define the set of nodes and hyperedges that will constitute it.

Nodes. The nodes of a code hypergraph are of two types: *code object* and (method) *trace*. A code object captures the notion of an object in an object-oriented program. A trace is more than just a basic block in a TCFG, it is a sequential set of instructions corresponding to an execution path for an entire method. For example, in Fig. 3, the trace node corresponding to the `middleButtonOnClick` method is composed of instructions that would span many basic blocks in a TCFG.

Definition 4 (Code Object). A code object v of type T in an object-oriented program P is an instantiated object variable of type T. For code object v of type T, we say $\mathsf{Datatype}\,(v) = T$.

Definition 5 (Trace). For a method M with entry instruction m_0 and set of exit instructions M_{exit}, a *method trace* is a path in a TCFG consisting of intra-procedural instructions from m_0 to m_{exit} for $m_{exit} \in M_{exit}$.

Hyperedges. There are two varieties of hyperedges we consider: one based on method invocations and the other based on objects being returned from non-void methods. Each *call hyperedge* is a many-to-one, annotated relationship among nodes in the hypergraph and is constructed for each method invocation. For a method invocation m in a method trace t, a hyperedge is constructed with the set of source nodes consisting of the node corresponding to t and the set of nodes corresponding to the formal parameter types of method m. The target of the hyperedge is a node corresponding to a method trace for method m. We annotate this node with the program state information for context as well as the method name. For a set of annotated hyperedges $E_{\mathcal{A}}$, $\mathsf{CallEdges}\,(E_{\mathcal{A}})$ defines the set of call hyperedges. Each *return hyperedge* is a one-to-one relationship between an origin trace node and a target code object with an annotation comprising of the method name for the origin node and program state information for context. For a program P, we say a *program state* σ of a program P is data store for all variables at a given execution point in P.

Definition 6 (Code Hypergraph). Let Π be the set of all program states for a program P with TCFG $\mathcal{T}$. A *code hypergraph* corresponding to a TCFG $\mathcal{T}$ is an annotated hypergraph $H\,(N, E_{\mathcal{A}})$ where, for each $n \in N$, n corresponds to either a (1) code object or (2) a method trace (acquired from an analysis of $\mathcal{T}$). Each directed hyperedge $e \in E_{\mathcal{A}}$ is defined by the ordered pair $e = (S, t, A)$ where $S \subseteq N$ and $t \in N$ is a target set of instructions corresponding to some method call. Each hyperedge annotation, $A \in \mathcal{A}$, is defined as a pair $A = (m, \sigma)$ where m is a method in the source code and $\sigma \in \Pi$. We say that a hyperedge (S, t, A) is labeled by m if $A = (m, \sigma)$ for some $\sigma \in \Pi$.

It is clear from Definition 6 that we can encode method invocations and returns as events and thus as annotations of hyperedges in a *code hypergraph*.

2.3 Sequence Diagrams

A sequence diagram is an instance of the more general *message sequence chart*. Succinctly, a message sequence chart [11, Ch. 4] [2,18] can be described as a set of partially-ordered, labeled events over a set of "processes". We will define a sequence diagram as a *code hyperpath* in a *code hypergraph*. A *code hyperpath* in a code hypergraph $H\,(N, E_{\mathcal{A}})$, constructed from a TCFG $\mathcal{T}$, is a hyperpath in H.

We now define a sequence diagram in terms of a (code) hyperpath in a *code hypergraph*.

Definition 7 (Hypergraph Sequence Diagram). Let $H\,(N, E_{\mathcal{A}})$ be a *code hypergraph*. Also let m be a method with entry point m_0 and let m_{exit} be an exit point of m. A *hypergraph sequence diagram* for method m corresponds to a hyperpath in H from the source hypernode of a hyperedge labeled m_0 to the target node of a hyperedge labeled m_{exit} and is denoted by $Y(H, m_0, m_{exit})$. The set of sequence diagrams $\mathcal{Y}(H, m_0, m_{exit})$ for a fixed pair of entry and exit points, m_0 and m_{exit} respectively, is the set of all $Y(H, m_0, m_{exit})$. Since a method has one fixed entry point and many possible exit points (given by M_{exit}), the collection of all such sequence diagrams (code hyperpaths) is given by $\mathscr{Y} = \bigcup_{m_{exit} \in M_{exit}} \mathcal{Y}(H, m_0, m_{exit})$.

We prove the equivalence of a message sequence chart with our notion of a sequence diagram as a hyperpath in [3].

To generate sequence diagrams, the *code hypergraph* is extracted according to the discussion in Sect. 3 where method m is a parameter specified by the user.

2.4 Characteristics of Sequence Diagrams

In this subsection, we formalize some properties of sequence diagrams that will be used by the query-based refinement interface for narrowing down the set of sequence diagrams generated to those that would be most "informative" to the user.

Depth of a Sequence Diagram. As a metric for code complexity, we define *depth* which relates the longest sequence of causally ordered messages without returning. We call $\mathcal{O}_1, \ldots, \mathcal{O}_n = \{\mathcal{O}\}_i$ an *object sequence* where for all $1 \leq i \leq n$, $\mathcal{O}_i$ are code objects. The *length* of the object sequence $\mathcal{O}_1, \ldots, \mathcal{O}_n$ is n. We define depth for a sequence diagram independent of the hypergraph definitions.

Definition 8 (Depth of a Sequence Diagram). The depth of a sequence diagram D is the greatest length d of the object sequence $\mathcal{O}_1, \ldots, \mathcal{O}_d$ in the diagram such that for each $1 \leq i \leq d-1$, there exists a message m_i from $\mathcal{O}_i$ to $\mathcal{O}_{i+1}$ and for each $1 \leq j \leq d-2$, $m_j \rightarrow m_{j+1}$ (m_j causally precedes m_{j+1}) and there does not exist any message m either from $\mathcal{O}_j$ to $\mathcal{O}_{j+1}$ or vice versa such that $m_j \rightarrow m$ and $m \rightarrow m_{j+1}$.

Interesting Sequence Diagrams. Not all sequence diagrams are of particular interest to a user. Requiring user interaction for refinement from the set of all sequence diagrams corresponding to a program is not ideal in terms of time and effort; therefore, we suggest a first step in formalizing the notion of an interesting sequence diagram to make interactions with *StaticGen* more efficient.

Formally defining an interesting sequence diagram requires quantification of some characteristic(s) of a sequence diagram. For a code hypergraph $H\,(N, E_{\mathcal{A}})$, we define function $\mathsf{Msg} : H \rightarrow \mathbb{N}$, as $\mathsf{Msg}\,(H) = |\mathsf{CallEdges}\,(E_{\mathcal{A}})|$. For a hypergraph sequence diagram D in $H\,(N, E_{\mathcal{A}})$, we define $\mathsf{Msg}(D) = \mathsf{Msg}(H)_D$ where the subscript denotes restriction to D and note that Msg is a measure that specifies the number of messages (method invocations) in the sequence diagram.

Let $\mathcal{D}_P$ be the set of all sequence diagrams for a program P. For $D \subseteq \mathcal{D}_P$, let $Msgs\,(D) = \{u \mid \exists D \in D \text{ s.t. } \mathsf{Msg}\,(D) = u \in \mathbb{N}\}$ and let $Msgs\,(D)_k$ denote the set of the k greatest elements of $Msgs\,(D)$ where $1 \leq k \leq |D|$. We define a function $\mathsf{select} : \mathbb{N} \rightarrow \mathcal{D}_P$ that, for a $u \in \mathbb{N}$, returns a sequence diagram $D \in \mathcal{D}_P$ such that $\mathsf{Msg}\,(D) = u$. If there exists multiple $D \in \mathcal{D}_P$ with $\mathsf{Msg}\,(D) = u$, ties are broken arbitrarily; $\mathsf{select}\,(u)$ is undefined if there does not exist any sequence diagram $D \in \mathcal{D}_P$ such that $\mathsf{Msg}\,(D) = u$. We define a function top that, for a set of sequence diagrams $\mathcal{D} \subseteq \mathcal{D}_P$ and a fixed number $0 \leq k \leq |\mathcal{D}|$, returns k sequence diagrams in $\mathcal{D}$ having the greatest number of messages. Formally, $\mathsf{top}\,(\mathcal{D}, k) = \{\mathsf{select}\,(u) \mid u \in Msgs\,(\mathcal{D})_k\}$ where $\mathcal{D} \subseteq \mathcal{D}_P$ and $1 \leq k \leq |\mathcal{D}|$.

Definition 9 (Interesting Sequence Diagram). For a program P with the set of sequence diagrams $\mathcal{D}_P$ with $|\mathcal{D}_P| = n$ and a fixed $0 < k \leq n$, D_P is an *interesting sequence diagram* if $D_P \in \mathsf{top}\,(\mathcal{D}_P, k)$.

For example, consider method `middleButtonOnClick` in Fig. 2 with generated set of sequence diagrams $\mathcal{D}$. The sequence diagram in Fig. 4 is interesting for $0 < k \leq 8$, since $\mathsf{Msg}\,(D) = 6$ for each $D \in \mathcal{D}$ describing a trace of `middleButtonOnClick`. The sequence diagram in Fig. 5 contains two messages and is therefore uninteresting for $0 < k \leq 8$.

While it is arguable that Definition 9 may not be ideal for every user, we believe that code complexity is often rooted in the number of method invocations and thus the probability is greater that a single trace can provide more information and thus is more likely to expose bugs and vulnerabilities.

3 Constructing the Hypergraph

In this section, we describe how *StaticGen* constructs a *code hypergraph* from an input set of code files; see [3] for pseudocode of the algorithms described here. The input to *StaticGen* is a set of (unobfuscated) code files in an object-oriented language. We assume that the code is processed by an intermediate system [15,29] into a TCFG. We construct a corresponding H and populate the nodes and hyperedges.

Nodes. As in Definition 6, there are two types of nodes in a *code hypergraph.* To construct both types of nodes, we parse the TCFG. For code objects, if a particular instruction is a declaration or a formal parameter, we add a corresponding node to H. If a node m defines a method prototype, we construct all possible traces for m using a process that identifies all possible naive execution paths for a method m over a control-flow graph; we then add each trace to the *code hypergraph.*

Hyperedges. We consider the two varieties of hyperedges in turn: *call hyperedge* and *return hyperedge.* A *call hyperedge* captures the callee trace, context of a caller through the set of input objects, and annotation of the method. For method calls, the hyperedge origin nodes consist of the callee trace nodes and the set of nodes corresponding to the actual parameters in the method call. The target node is then a node corresponding to a trace of the called method. The result is a call hyperedge for the *code hypergraph* with an annotation consisting of the name of the called method and an empty program state. For non-void methods, we construct a *return hyperedge* relating the current trace node as source and the object being returned as target annotated with the called method name and empty program state and an indicator that it is a return hyperedge.

4 Static Sequence Diagram Construction

Sequence diagram generation consists of three phases: (1) sub-hypergraph identification through pebbling [8], (2) hyperpath identification, and (3) converting from a hyperpath to a sequence diagram. For a more detailed description and pseudocode of each phase, see the extended paper [3].

Pebbling. Pebbling is a linear-time traversal over an annotated hypergraph that identifies a sub-hypergraph [6, Ch. 1] satisfying constraints placed on code objects and methods by the user. Pebbling is a breadth-first traversal over an annotated hypergraph where we mark each node with a pebble once it is visited using the rule "once all source nodes in a hyperedge have been pebbled, the target node is pebbled" (similar to the Dowling and Gallier [8] marking algorithm for satisfiability of propositional horn clauses). For example, in Fig. 3, if we pebble the trace node for `middleButtonOnClick`, we immediately pebble the trace node for `getBlue`. In turn, we pebble the code object node for `c`. Then, since both source nodes are pebbled, we pebble the target trace node for `SetUpperRightButton`. We assume that a *code hypergraph* has been pebbled resulting in a *pebbled code hypergraph.*

Hyperpath Identification. For a given method m, we construct the corresponding set of all hyperpaths in a *code hypergraph* H. Our algorithm maintains the same information for a sequence diagram as stated in Definition 7, but instead of maintaining a code hyperpath the result is an equivalent path consisting of one-to-one edges and an associated set of objects. For simplicity, we consider constructing a single path $\mathcal{P}$ by considering a single trace T of m. Recall that a trace T is a sequential set of instructions. For each instruction $i \in T$, we consider if i is a method invocation. If i is not a method invocation, we add i to $\mathcal{P}$ maintaining sequential order of instructions. If i is a method invocation, we recur with a trace of the method called in i. $\mathcal{P}$ is then a valid, sequential ordering of instructions for method m.

Hyperpath to Sequence Diagram Conversion. Given a path $\mathcal{P}$ corresponding to a hyperpath in a *code hypergraph*, we construct the corresponding sequence diagram D. For each method invocation instruction i in $\mathcal{P}$, we add to D each of the following: (a) the invocation of method m in i as a message, (b) a recursively constructed sub-sequence diagram of m, and (c) a message indicating a return from m.

5 Interface for Diagram Generation

A sequence diagram D has features such as: depth as defined in Definition 8, number of messages (number of call hyperedges), types of all code objects, method coverage, and branch coverage. In this section, we describe the query language, the interface for query-based refinement, and provide some examples.

5.1 Query over the Language of Sequence Diagrams

We define a query over the language of sequence diagrams. The language of sequence diagrams $\mathcal{L}$ is defined over the alphabet Σ consisting of code objects and method traces. For simplicity, we will refer to code objects as c_i with $i \in \mathbb{Z}^+$, method traces as m_j with $j \in \mathbb{Z}^+$ with corresponding method returns m'_j. Hence, $\Sigma = \{c\}_i \cup \{m\}_j \cup \{m\}'_j$ for i and j finite in $\mathbb{Z}^+$ and $i \geq 1$ and $j \geq 1$.

We note that a hyperpath Y in a *code hypergraph* H is a string in $\mathcal{L}$ [3] since a topological sort of the DAG corresponding to Y results in a string $s \in \mathcal{L}$. We note that distinct orders of topological sorts on a DAG corresponding to a hyperpath will result in distinct strings; however, each such string is unique in $\mathcal{L}$ over the original program. A query is defined over a set of sequence diagrams $\mathcal{D} \subseteq \mathcal{L}$ generated using the techniques described in Sect. 4; however, generation can be more targeted. It is often cumbersome and unnecessary to generate all sequence diagrams beginning at a `main` method in a program. Generation can be performed on-demand beginning at any method reducing the size of the corresponding hypergraph. In order to acquire the initial set of sequence diagrams D_S, we may use the predicate "`start` M," where M is a method dictating where the resultant sequence diagram(s) will begin.

Query Operations. A query $Q = \{q\}_i$ over $\mathcal{L}$ consists of a finite sequence of operations $\{q\}_i$ that *refine* the given set of sequence diagrams $\mathcal{D} \subseteq \mathcal{L}$ into the resulting set $Q(\mathcal{D}) = F \subseteq \mathcal{D}$.

- For a method trace $\ell \in \Sigma$, "`filter` ℓ $\mathcal{D}$" prunes the substring from ℓ to ℓ' in each sequence diagram in $\mathcal{D}$. This removal process efficiently eliminates calls to library-based functionality or method definitions that are not of interest.

For a set of code objects $\ell \subseteq \Sigma$, "`filter` ℓ $\mathcal{D}$" prunes all characters $c \in \ell$ from each string in $\mathcal{D}$. Removal of a code object allows the user to refine the set of sequence diagrams by omitting specific variables.

- For a set of predicates R describing strings in $\mathcal{L}$, "`remove` R $\mathcal{D}$" will remove all resulting sequence diagrams for which all $r \in R$ evaluate to `true`. The complementary operation "`accept` R $\mathcal{D}$" will collect all sequence diagrams for which all $r \in R$ evaluate to `true`.
- For an integer k, "`top-interesting` k $\mathcal{D}$" returns `top` $(\mathcal{D}, k)$.
- "`meth-cover` p $\mathcal{D}$" and "`br-cover` p $\mathcal{D}$" each return sequence diagrams ensuring minimal method and branch coverage respectively for a lower bound percentage p ([3] formally defines method and branch coverage).

We define a simple grammar for a query Q over $\mathcal{L}$; the terminal symbols include ℓ, R, $0 \leq p \leq 1$, and $k \in \mathbb{Z}^+$ as defined above.

$$\begin{aligned} Q(\mathcal{D}) \rightarrow \mathcal{D} &\mid \texttt{filter}\ \ell\ Q(\mathcal{D}) \mid \texttt{remove}\ R\ Q(\mathcal{D}) \mid \texttt{accept}\ R\ Q(\mathcal{D}) \\ &\mid \texttt{top-interesting}\ k\ Q(\mathcal{D}) \mid \texttt{meth-cover}\ p\ Q(\mathcal{D}) \\ &\mid \texttt{br-cover}\ p\ Q(\mathcal{D}) \end{aligned}$$

5.2 Query Interface to Diagram Generation

We present an interface where a user of *StaticGen* can query over the set of sequence diagram features to obtain a subset of sequence diagrams. Our methodology requires manual input of the code as well as a query Q as previously described. Depending on the specification of Q, we may omit, through the pebbling process, call hyperedges corresponding to method calls that may be removed. Given a pebbled code hypergraph, we construct the corresponding set of all sequence diagrams. We then `filter` the resulting set of sequence diagrams related to method removal, coverage, or `top` into the desired set of sequence diagrams.

If the user wishes to refine Q into Q' we may re-pebble the code hypergraph and generate according to Q'. Our query system provides continual refinement until the appropriate set of sequence diagrams is acquired. That is, initially, a user might simply request a set of interesting sequence diagrams. Then, as the user becomes more familiar with the code base, they may define a more restricted query. This process of query refinement can continue ad nauseum.

Within the bounds of the user selected query, we prioritize what the user sees by first eliminating strictly isomorphic diagrams [3] and diagrams which are "subsets" of other diagrams. We then determine the set of sequence diagrams S that match the user's query. Using a method coverage metric for the code, we prioritize the diagrams into a list I.

5.3 Sample Queries

Assume the user specifies as input the code base containing the source code in Fig. 2. To filter elements from the set of resulting sequence diagrams, the user defines a query Q with `start` being `middleButtonOnClick` and filters object `r` and its corresponding methods as well as the `setText` method. The result is eight diagrams, seven of which are strictly isomorphically unique [3], and one of which is shown in Fig. 4. If we append to Q an `accept` predicate with method `SetUpperRightButton(int)`, the only diagram returned is shown in Fig. 4.

As another example, Fig. 5 arises from a query requesting the least interesting diagram from analyzing all methods.

6 Experimental Results

Timely generation of sequence diagrams depends on two factors: (1) complexity of branching in the given code and (2) user-defined queries to pebble the hypergraph and prune the resultant set sequence diagrams. For our experiments, we limit diagram generation to package prefixes. This limitation allows the user to visualize internal package interactions without dealing with bloat from exterior execution paths to that package. We ran our generation algorithm on a desktop with Intel Core i5-4460 at 3.2 GHz with 8 GB RAM on 64-bit Linux Mint operating system.

Benchmark Code. Our initial tests have focused on open-source Android bytecode applications taken from [1,35] with wide-ranging focus, including: ad blocking, email, and web browsing. The bytecode was input into the Soot framework [29] which can process bytecode or source code thus bringing the same capabilities to bear, independent of input format. [3] lists the projects and corresponding facts about each code base in the chosen corpus, including the package we analyzed, the number of constituent classes, processing time, and the operation count.

Soot analyzes bytecode by breaking down classes into groups of methods, and methods into groups of abstract statements; the number of abstract statements is referred to as the operation count. While operation count may not correspond one to one with source lines of code, it does correspond to essential logical statements executed by the processor, and is a useful measurement of the complexity of the program analyzed. The operation count for our corpus is shown in [3].

As another measure of complexity of the target Android code, we consider the histogram in Fig. 6 depicting the mean depth of diagrams for non-library functionality for each benchmark Android package. Our event-driven benchmarks are generally shallow as is evident in Fig. 6; the mean depth among all packages is 1.29 with standard deviation 0.92. We view the depth metric as a guide to the number of corresponding sequence diagrams; the greater the depth, the more diagrams should result. Figure 7 is a scatterplot of the relationship between mean depth and number of diagrams generated. We see a linear model given by $y = 887.58x - 496.56$, where y is the number of diagrams generated, and x is the mean depth of the set of diagrams for an Android package. The correlation is moderate with correlation coefficient $r^2 = 0.5643$.

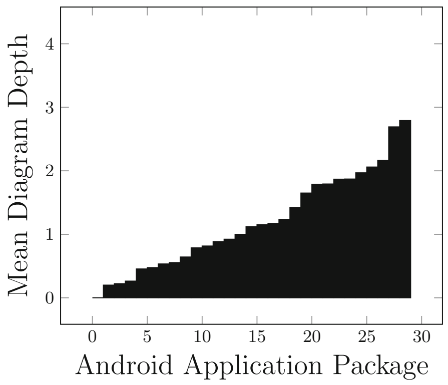

Fig. 6. Mean sequence diagram depth per android application package

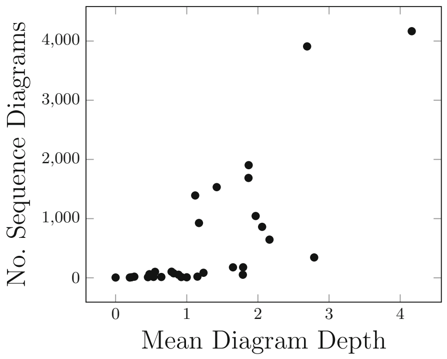

Fig. 7. No. generated sequence diagrams vs. mean diagram depth for the entire corpus

Time and Scope of Synthesis. We measure tool efficiency by considering generation time. Our reported execution times include Soot's Simplification [29] procedure, hypergraph construction, diagram generation, and refinement. In Fig. 8, several Android packages are processed quickly. However, the mean of 96.78s and the standard deviation of 174.58s indicates more complex packages result in greater time dispersion. For each Android package, Fig. 9 describes the number of diagrams that give complete method coverage. Some of the more complex packages skew the distribution (std. dev. 1085.23 and mean 647.1) with a strongly correlated linear model ($r^2 = 0.9082$) comparing the number of diagrams with respect to generation time. This is strong evidence indicating our technique does not require a significant amount of processing time for code bases with large sets of sequence diagrams.

Comparison with Dynamic Synthesis Tools. Several existing tools for sequence diagram generation are based on traces saved from debug runs of a program [10, 24]. While our approach differs significantly in that we utilize static analysis to generate diagrams from a large number of potential execution paths, we did compare our approach against Diver [24]. We found that *StaticGen* was able to construct similar sequence diagrams compared to Diver which uses a dynamic approach. The full text of this comparison is available at [3].

Evaluation of Interestingness. We test the usefulness of our interestingness metric by examining the possibility of using it in uncovering *security vulnerabilities in code*. We selected an independently studied example for assessing our definition of interestingness. Both Livshits [19] and Sampaio [31] used a web application named BlueBlog [7] in their corpora of applications with security vulnerabilities. In addition, [31] provided a software tool, ESVD [30], to analyze code for vulnerabilities.

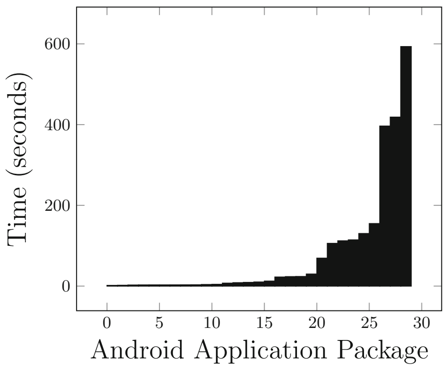

Fig. 8. Time per android application package

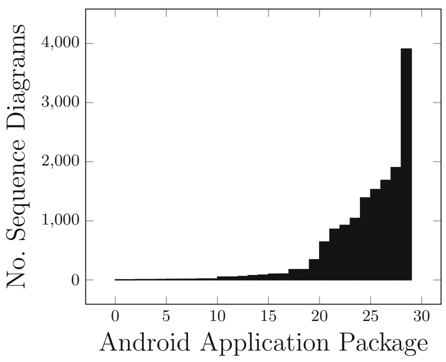

Fig. 9. Number of sequence diagrams per android application package

We focus on a vulnerability evident in the unsafe http request in the code in Fig. 10 that was originally detected by ESVD [30]. The value returned by the `getServletPath` function is stored in the variable `url` and is not sanitized by both branches. Without any filtering, *StaticGen* generated 2800 diagrams from BlueBlog, while the interestingness criterion narrowed it down to 45 diagrams ranked according to `top` defined in Sect. 5. The `doGet` method was the subject of the diagrams ranked 22 and 32 of the 45 interesting diagrams; a fragment of the rank-22 diagram is shown in Fig. 11. This example shows that prioritizing novel information allows us to reduce a set of diagrams to a useful fraction, while retaining information about crucial code paths, that can then be delegated to a human expert or a vulnerability analysis tool for further analysis for security vulnerabilities. It is possible that an excluded diagram may contain the vulnerability; however, our query focusing on interestingness, by definition, includes diagrams that provide novel information. Hence, a vulnerability as we have described cannot hide only in the discarded diagrams.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException
{
   String url = request.getServletPath() + toPath;
   if( forward ) {
      RequestDispatcher disp = getServletContext().getRequestDispatcher(url);
      disp.forward(request, response);
   } else {
      response.sendRedirect(url);
   }
}
```

Fig. 10. BlueBlog [7] doGet function

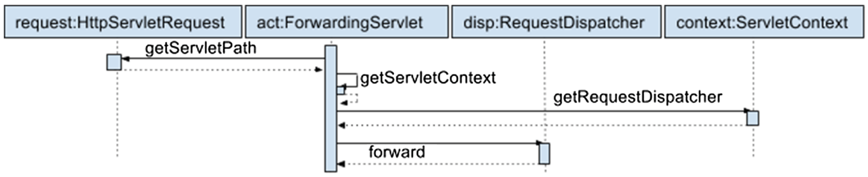

Fig. 11. BlueBlog [7] doGet sequence diagram fragment

7 Related Work

In [16,17,22], Lo, et al., propose techniques for dynamic specification mining by inferring sequence diagrams over execution traces that include inter-object behavior and causal ordering. Lo, et al. use a graph of symbolic message sequence charts as an intermediate representation while we invoke a hypergraph representation. Tools such as jTracert [5] and Object-Aid [25] generate sequence diagrams directly from application runtime while [25] uses the Eclipse IDE [9] to reverse engineer all or part of a stack trace. Similarly, [12] divides a long dynamic trace of a Java program into a series of smaller diagrams culminating in a sequence diagram. Finally, [36] describes an approach for generating sequence diagrams dynamically using a k-tail merging algorithm that merges the collected traces. The goal of merging by [36] is to construct a single sequence diagram. Our technique does not limit generation to a single diagram, but generates a complete space of sequence diagrams that is refined by query.

There are several tools that statically generate sequence diagrams. Visual paradigm [34] is a simple tool for sequence diagram generation that is in one-to-one correspondence with the source code without refinement. Other tools such as eUML2 Modeler [32] and Visual Studio [23] generate diagrams statically, but also offer the ability for the user to refine the diagram by selection or omission of methods. Similarly, Architexa [4] generates sequence diagrams, but is completely interactive with the user during construction. While all of these tools are based on a static analysis of the target code, none of these tools automate the refinement process based on a query scheme over the set of all possible diagrams.

The Interaction Scenario Visualizer (ISVis) [13], employs a combination of static and trace-based information and communicates the overall importance of visualizing source code. Tonella and Potrich [33] describe static extraction of UML sequence diagrams from C++ code using partial analysis and focusing, but do not perform analysis of intraprocedural flow of control. The CPP2XMI tool [14] processes XMI into sequence diagrams with no means of user-based refinement as with *StaticGen*. I2SD [26] is a static generation tool that leverages metadata through interceptors whereas our technique does not rely on such information. The RED tool [27,28] was a significant step forward in reverse-engineering diagrams by mapping reducible CFGs to interactions. In contrast, our use of an annotated hypergraph provides the means to refine the object interactions, context, and causal ordering based on user query; in some respects, our approach attempts to fill the "exploration mode" described in [27]. In total, our approach seeks to empower the user by supporting query-based refinement over the set of all sequence diagrams.

In [20,21] authors present techniques for *user-guided specification mining* over executions traces by proposing approaches to filter mined sequence diagrams. We similarly aim to support property discovery through an iterative and interactive approach by incorporating a notion of interestingness.

8 Conclusions

This paper describes a framework for static generation of sequence diagrams using a directed hypergraph to encode message context, interactions, and causality. Based on a user-query, we prune the sequence diagram space through a pebbling procedure to generate the desired set of sequence diagrams. We showed that, in practice, our framework provides the basis for interactive software archeology as well as an important tool for debugging legacy code.

References

1. List of Open-Source Android Apps (2013). http://forum.xda-developers.com/showthread.php?t=2124002
2. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. In: ICSE 2000, pp. 304–313 (2000). http://doi.acm.org/10.1145/337180.337215
3. Alvin, C., Peterson, B., Mukhopadhyay, S.: StaticGen: Static Generation of UML Sequence Diagrams - Technical Report (2017). http://hilltop.bradley.edu/~calvin/papers/fase17-technical.pdf
4. Architexa.com: Introduction to Architexa—Sequence Diagram Generation (2015). http://www.architexa.com/support/videos/sequence-diagrams
5. Bedrin, D.: jtracert (2015). https://code.google.com/p/jtracert/
6. Berge, C.: Graphs and Hypergraphs, vol. 45. North-Holland Mathematical Library, Elsevier Science Publishers B.V. (1989)
7. Burén, R.: BlueBlog. https://sourceforge.net/projects/blueblog/. Accessed 16 Oct 2016
8. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. J. Log. Program. **1**(3), 267–284 (1984). dx.doi.org/10.1016/0743-1066(84)90014–1
9. Eclipse Foundation Inc: Eclipse (2015). https://eclipse.org/
10. Gestwicki, P.V., Jayaraman, B.: JIVE: Java interactive visualization environment. In: OOPSLA 2004, pp. 226–228. ACM, New York (2004). http://doi.acm.org/10.1145/1028664.1028762
11. Harel, D., Thiagarajan, P.: Message sequence charts. In: Lavagno, L., Martin, G., Selic, B. (eds.) UML for Real: Design of Embedded Real-time Systems, 1st edn. Kluwer Academic Publishers (2003)
12. Ishio, T., Watanabe, Y., Inoue, K.: AMIDA: A sequence diagram extraction toolkit supporting automatic phase detection. In: ICSE 2008, pp. 969–970 (2008). http://doi.acm.org/10.1145/1370175.1370212

13. Jerding, D.F., Stasko, J.T., Ball, T.: Visualizing interactions in program executions. In: ICSE 1997, pp. 360–370 (1997). http://doi.acm.org/10.1145/253228.253356
14. Korshunova, E., Petkovic, M., van den Brand, M.G.J., Mousavi, M.R.: CPP2XMI: Reverse engineering of UML class, sequence, and activity diagrams from C++ source code. In: WCRE 2006, pp. 297–298 (2006). http://dx.doi.org/10.1109/WCRE.2006.21
15. Kroening, D.: goto-cc–A C/C++ Front-End for Verification (2015). http://www.cprover.org/goto-cc/
16. Kumar, S., Khoo, S., Roychoudhury, A., Lo, D.: Mining message sequence graphs. In: ICSE 2011, pp. 91–100 (2011). http://doi.acm.org/10.1145/1985793.1985807
17. Kumar, S., Khoo, S., Roychoudhury, A., Lo, D.: Inferring class level specifications for distributed systems. In: ICSE 2012, pp. 914–924 (2012). http://dx.doi.org/10.1109/ICSE.2012.6227128
18. Leucker, M., Madhusudan, P., Mukhopadhyay, S.: Dynamic message sequence charts. In: Agrawal, M., Seth, A. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 253–264. Springer, Heidelberg (2002). doi:10.1007/3-540-36206-1_23
19. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: SSYM 2005, pp. 18–18. USENIX Association, Berkeley (2005). http://dl.acm.org/citation.cfm?id=1251398.1251416
20. Lo, D., Maoz, S.: Mining scenario-based triggers and effects. In: ASE 2008, pp. 109–118 (2008). http://dx.doi.org/10.1109/ASE.2008.21
21. Lo, D., Maoz, S.: Mining hierarchical scenario-based specifications. In: ASE 2009, pp. 359–370 (2009). http://dx.doi.org/10.1109/ASE.2009.19
22. Lo, D., Maoz, S., Khoo, S.: Mining modal scenario-based specifications from execution traces of reactive systems. In: ASE 2007, pp. 465–468 (2007). http://doi.acm.org/10.1145/1321631.1321710
23. Msdn.microsoft.com: Visualize Code on Sequence Diagrams (2015). https://msdn.microsoft.com/en-us/library/ee317485.aspx
24. Myers, D., Storey, M.A.: Using dynamic analysis to create trace-focused user interfaces for IDEs. In: FSE 2010, pp. 367–368. ACM, New York (2010). http://doi.acm.org/10.1145/1882291.1882351
25. Objectaid.com: UML Explorer (2015). http://www.objectaid.com/sequence-diagram
26. Roubtsov, S.A., Serebrenik, A., Mazoyer, A., van den Brand, M.G.J., Roubtsova, E.E.: I2SD: Reverse engineering sequence diagrams from enterprise java beans with interceptors. IET Softw. 7(3) (2013). http://dx.doi.org/10.1049/iet-sen.2012.0056
27. Rountev, A., Connell, B.H.: Object naming analysis for reverse-engineered sequence diagrams. In: ICSE 2005, pp. 254–263 (2005). http://doi.acm.org/10.1145/1062455.1062510
28. Rountev, A., Volgin, O., Reddoch, M.: Static control-flow analysis for reverse engineering of UML sequence diagrams. In: PASTE 2005, pp. 96–102 (2005). http://doi.acm.org/10.1145/1108792.1108816
29. Sable Research Group: Soot: A framework for Analyzing and Transforming Java and Android Applications (2015). http://sable.github.io/soot/
30. Sampaio, L.: Early Security Vulnerability Detector. https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd. Accessed 16 Oct 2016
31. Sampaio, L., Garcia, A.: Exploring context-sensitive data flow analysis for early vulnerability detection. J. Syst. Softw. **113**, 337–361 (2016). http://www.sciencedirect.com/science/article/pii/S0164121215002873

32. Soyatec.com: Soyatec - Sequence diagram generation (2015). http://www.soyatec.com/euml2/features/eUML2%20Modeler/
33. Tonella, P., Potrich, A.: Reverse engineering of the interaction diagrams from C++ code. In: ICSM 2003, pp. 159–168 (2003). http://dx.doi.org/10.1109/ICSM.2003.1235418
34. Visual-paradigm.com: Reverse Engineering Sequence Diagram from Java Source Code (2015). https://www.visual-paradigm.com/tutorials/seqrev.jsp
35. Wikipedia: List of Free and Open-Source Android Applications (2015). http://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications
36. Ziadi, T., da Silva, M.A.A., Hillah, L., Ziane, M.: A fully dynamic approach to the reverse engineering of UML sequence diagrams. In: ICECCS 2011, pp. 107–116 (2011). http://dx.doi.org/10.1109/ICECCS.2011.18