International Journal on Software Tools for Technology Transfer (2021) 23:31–53
https://doi.org/10.1007/s10009-019-00545-z

GENERAL

Regular Paper

Static generation of UML sequence diagrams

Chris Alvin[1] · Brian Peterson[2] · Supratik Mukhopadhyay[2]

Published online: 26 October 2019

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* (supporting software: *StaticGen*), analyzing a corpus of 30 Android applications. We provide experimental results demonstrating the efficacy of our technique (originally appeared in the Proceedings of Fundamental Approaches to Software Engineering—20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017).

Keywords Static analysis · Scenario diagram generation · Sequence diagram · Code hypergraph · Query language · Vulnerability analysis

1 Introduction

Legacy object-oriented code may be accompanied by high-level documentation and/or descriptive comments in the source code, 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 [5,22,23,29,34] can synthesize a subset of all possible sequence diagrams based on runtime traces. The capability of purely dynamic reverse-engineering techniques to produce useful diagrams depends on the quality of the executions. In particular, one may need a large number of executions with sufficient diversity to cover the space of interactions. Existing static techniques [43] result in sequence diagrams that replicate the original legacy source code, including conditionals and loops, without providing further intuitive notions beyond the code itself. Hybrid techniques like [18,42] combine static and dynamic analysis. Information extracted from an accurate static analysis framework can guide the executions during the dynamic stage.

We present a technique, depicted in Fig. 1, for static generation of UML sequence diagrams[1] together with a query system to guide the user to the most interesting interactions in the (unobfuscated) source code[2] Given an existing object-

✉ Chris Alvin
ctalvin@gmail.com; chris.alvin@furman.edu

Supratik Mukhopadhyay
supratik@csc.lsu.edu

[1] Furman University, 3300 Poinsett Highway, Greenville, SC 29613, USA

[2] Louisiana State University, 102F Electrical Engineering Building, Baton Rouge, LA 70803, USA

[1] Some authors [20,24] refer to these diagrams as scenario diagrams. This is a segue to UML sequence diagrams.

[2] Our technique will work on obfuscated code as well, but the resulting sequence diagrams may be difficult to understand.

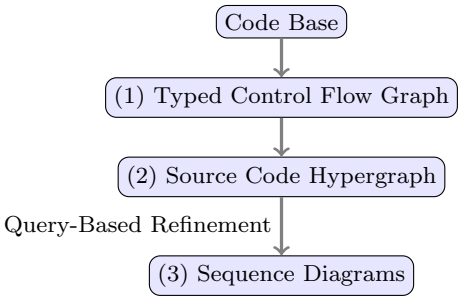

Fig. 1 The *StaticGen* system flowchart

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 graph annotated with type information—a familiar structure in static analysis acquired from an existing front-end tool such as Soot [39] or goto-cc [21].

The TCFG for a program P captures the execution of P, but does not explicitly capture (a) the interactions among the objects constituting P, (b) their context, and (c) the *causal ordering* of their interactions. For example, the TCFG in Fig. 3 does not explicitly capture the fact that a call to `middleButtonOnClick` requires the context of an object of type `View`. Similarly, it does not explicitly capture the fact that the operations `setText("clicked"), opt = r.nextInt(), c = getBlue()`, and `SetUpperRightButton(c)` take place within the method `middleButtonOnClick`. In addition, the TCFG does not have any explicit construct to represent interactions among objects. A TCFG does not also contain the complete context of object information. That is, the TCFG contains the static datatype of a particular object, but the TCFG does not explicitly maintain the superclass hierarchy for each object.

For the code in Fig. 2, the TCFG is depicted in Fig. 3. Hence, the second step of our methodology involves constructing a directed *code hypergraph* [7] (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 [7]) in Fig. 4. 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. 4). The second category of nodes, called *trace* nodes, capture a trace of a method (rectangles in Fig. 4). For example, it is clear that `middleButtonOnClick` in Fig. 2 has $2 * 4 = 8$ possible traces due to the permutation of respective branches; Fig. 4 depicts one of those 8 trace nodes.

```
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

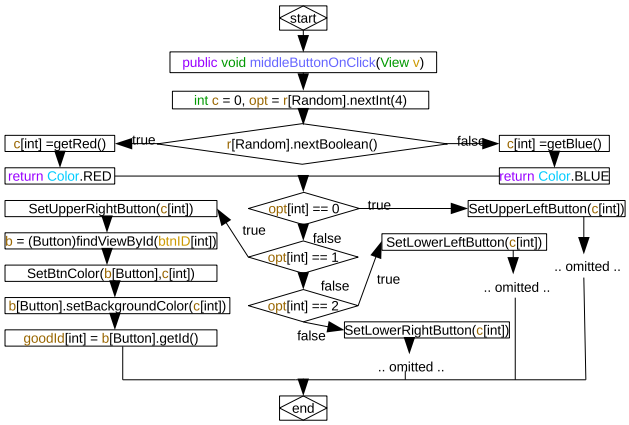

Fig. 3 The typed control-flow graph for Fig. 2

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. In Fig. 2, for example, a call to `middleButtonOnClick` requires the context of an object of type `View` and a callee; the corresponding hyperedge in Fig. 4 is labeled accordingly with the destination method and program state information for context.

The third step in our technique (depicted in Fig. 1) constructs sequence diagrams (Sect. 5) given a *code hypergraph* corresponding to an input code base. Each hyperpath [7] in

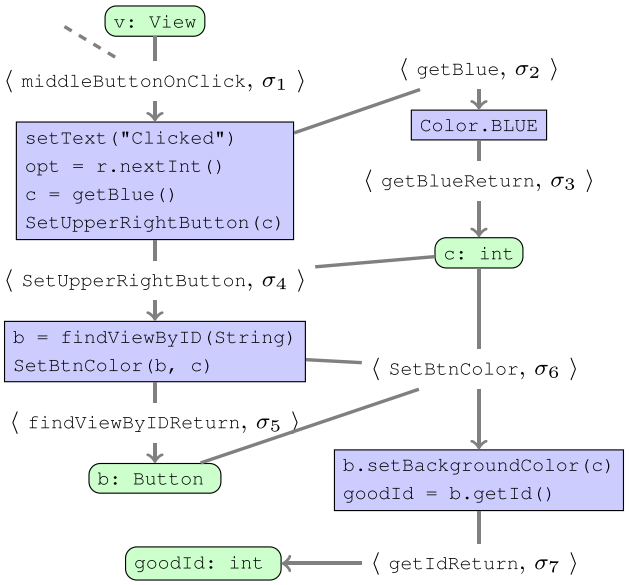

Fig. 4 A portion of the code hypergraph corresponding to the code in Fig. 2

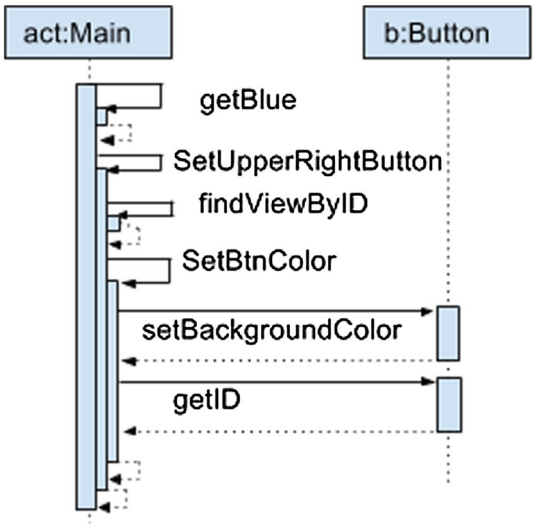

Fig. 5 Sequence diagram for an execution path in Fig. 2

Fig. 6 Uninteresting sequence diagram for Fig. 2

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. 4 corresponds to the sequence diagram shown in Fig. 5. 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 [12,44]. *StaticGen* generated 647.1 sequence diagrams on average per package taking a mean of 96.78 seconds 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.

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. 5) 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. 6).
- We illustrate the efficacy of our technique (Sect. 7) 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. When a loop is encountered in a TCFG, we treat the loop as a conditional branch by taking it once; it is out of scope of this work to consider other methods for handling loops (e.g., loop summarization, ranking functions, etc.) 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* [7] 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* [1]; 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. For example, the `middleButtonOnClick` method takes a parameter of object `v` of type `View` and is represented as a code object in the corresponding code hypergraph in Fig. 4 (indicated by rounded edges). 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. 4, a trace node corresponding to one path of the `middleButtonOnClick` method is composed of instructions that would span many basic blocks in a TCFG. That is, a complete code hypergraph for the source code in Fig. 2 would contain 8 distinct trace nodes for the `middleButtonOnClick` method consistent with the 8 unique execution paths.

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 $\mathsf{Datatype}(v) = T$.

The applications of our theory rely upon the capabilities of a front-end tool. In this paper, we deferred to Soot [39] to provide datatype information from polymorphic code and dynamic bindings; Soot provides a query mechanism to identify the datatype of a particular object. The definition of code object in Definition 4 captures the fact that our technique relies on using a tool that provides a query mechanism for static object type information. Thus, the quality of the proposed procedures would suffer if the underlying analyzer is lacking.

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 $m_{exit} \in M_{exit}$.

With each node in a *code hypergraph* we associate one of two categories $Cats = \{\mathsf{object}, \mathsf{trace}\}$; we define a corresponding function cat that maps a node to its category. We may then define a function trace to extract

the trace nodes from a set of nodes N, $\mathsf{trace}(N) = \{n \mid n \in N \wedge \mathsf{cat}(n) = \mathsf{trace}\}$.

Hyperedges There are two varieties of hyperedges we consider: *call hyperedges* based on method invocations and *return hyperedges* that represent objects being returned from non-void methods; our data-driven approach does not require return edges for void methods as edges enforce the dependence between control and information (objects).

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.

As an example, Fig. 4 depicts a call hyperedge with source nodes consisting of two code objects (`b` and `c`) and a trace node representing the execution path of `SetUpperRightButton`. The target node of this call hyperedge is a trace node for `SetBtnColor`. In general, a hyperedge consists of $n + 1$ source nodes where n is the number of formal parameters as well as one target node. Finally, the edge is annotated with the name of the method being invoked (`SetBtnColor`) as well as state information (σ_6). 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.

Each *return hyperedge* is a one-to-one relationship between an origin trace node and a target code object with an annotation comprised of the method name for the origin node and program state information for context. There are two return hyperedges in Fig. 4 representing code objects being returned. Specifically, `getBlue` returns a value which will be assigned to variable `c` (in `middleButtonOnClick`) indicated with a dotted line and annotated with `getBlueReturn` and state information (σ_3).

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*. The literature (e.g., [1,14,25]) defines a *message sequence chart* over a set of processes and an alphabet as a tuple consisting of:

- a partitioned set of "send" and "receive" events,
- a mapping that maps an event to a process,
- a bijective mapping that matches a send message with a corresponding receive message,
- a mapping that labels each event, and
- a partial order on the set of events.

Succinctly, a message sequence chart [1,14,25] 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.

Lemma 1 *Let $H(N, E_{\mathcal{A}})$ be a code hypergraph corresponding to an acyclic TCFG $\mathcal{T}$, then a sequential hyperpath Y in H adheres to the sequential order of instructions induced by $\mathcal{T}$.*

Proof Suppose Y does not adhere to the sequential order of instructions induced by the TCFG $\mathcal{T}$. We consider two distinct cases.

Assume there exists two instructions i_1 and i_2 in a single method in $\mathcal{T}$ such that i_1 is a predecessor of i_2 in $\mathcal{T}$, but i_1 is not a predecessor of i_2 in Y. In this case, both i_1 and i_2 are instructions in a single method trace node in Y and i_1 will precede i_2 by definition of a method trace. This is a contradiction.

In the second case, suppose there exists two instructions i_1 in method m_1 and i_2 in method m_2 in $\mathcal{T}$ such that i_1 is a predecessor of i_2 and in Y i_1 is not a predecessor of i_2. In this case, i_1 and i_2 are instructions that are contained in two distinct method trace nodes in Y. That is, i_1 is an instruction in method trace node m_1 for method f_1 in Y; similarly for i_2 in node m_2 describing method f_2. It must be the case that i_1 is a method invocation or must precede a method invocation of f_2 from f_1. The definition of *code hypergraph* requires that there exists a directed hyperedge $e = (S, t, A)$ in Y where $m_1 \in S$ and $t = m_2$. It is now the case that, assuming all code objects $S \setminus \{m_1\}$ have been initialized appropriately, i_1 will precede (or invoke directly) the call to method f_2 executing instructions, including i_2. This is a contradiction. □

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.

Lemma 2 *A hypergraph sequence diagram is a message sequence chart.*

Proof Let $H\,(N, E_{\mathcal{A}})$ be a *code hypergraph*. Also let m be a method with entry point m_0 to exit point m_{exit} in H. Let $Y(H, m_0, m_{exit})$ be a hyperpath sequence diagram in H from m_0 to m_{exit}. We verify Y satisfies the criteria for a message sequence chart.

(1) *A partitioned set of "send" and "receive" events.* As stated in Definition 6, a code hypergraph consists of directed hyperedges that are annotated with method invocations and return statements as events. The set of "send" events S corresponds to method call invocation hyperedges in Y while the set of "receive" events R is the set of hyperedges corresponding to method returns in Y. In Y, each method invocation has a corresponding return; therefore, $|S| = |R|$.

(2) *A mapping that maps an event (method invocations) to a process (object).* Each hyperedge in Y consists of two objects: the currently active object and the callee object. It is clear that a unique mapping exists that takes a hyperedge and returns the source object; similarly for the callee object.

(3) *A bijective mapping that matches a send message with a corresponding receive message.* Each method invocation is encoded as a hyperedge in Y. Take a method invocation between m_0 and m_{exit} with method entry m'_0 and method exit m'_{exit}; note $m_0 \Rightarrow m'_0 \Rightarrow m'_{exit} \Rightarrow m_{exit}$. Then, in order for Y to be valid, each such sub-hyperpath $Y_s(H, m'_0, m'_{exit})$ must also be valid subhyperpaths in H. That is, there exists a hyperedge in H corresponding to the method invocation as well as a corresponding method return. It is clear that such a bijection exists for mapping a sequential hyperpath method invocation to a return in Y.

(4) *A mapping that labels each event.* It is clear that each hyperedge encodes a method invocation as an event in Y; hence, a mapping exists that will label the set of events in Y.

(5) *A partial order on the set of events.* It is clear that the ordering of events encoded as hyperedges in Y are reflexive and antisymmetric. Let $e_A = (S_A, t_A, A), e_B = (S_B, t_B, B)$, and $e_C = (S_C, t_C, C)$ be hyperedges in Y such that, without loss of generality, $A \rightarrow B$ and $B \rightarrow C$. By assumption, $t_A \in S_B$ and $t_B \in S_C$. That is, we are guaranteed to invoke the instructions defined by nodes t_A, t_B, and t_C in that order. That is, the instructions in t_C will be invoked only after the instructions in t_A. Hence, the partial order is maintained from the message invoked by A through message B, and last through message C. □

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 generated set of sequence diagrams to those sequence diagrams that would be most "informative" to the user.

Since it may be beneficial to structurally compare two sequence diagrams, we define analogy of sequence diagrams through causal ordering of method calls.

We use the term *analogous* to define a sequence diagram S that is equivalent to a sequence diagram T in terms of causal ordering. We formalize the notion of coarse and strictly analogous sequence diagrams by viewing sequence diagram hyperpaths as graphs.

Definition 8 (*Coarse Sequence Diagram Homomorphism*) For code hypergraphs $H\,(N, E_{\mathcal{A}})$ and $H'\,(N', E_{\mathcal{A}}{}')$, $\phi : H \rightarrow H'$ is a *coarse sequence diagram homomorphism* if for $1 \leq i \leq k$ and $v, v_i \in N$, for all $\langle v_1, \ldots, v_k \rangle = \mathbf{v} \in 2^V$ such that $(\mathbf{v} \rightarrow v) \in E_{\mathcal{A}}$, v and $\phi(v)$ are nodes in which $\mathsf{cat}\,(v) = \mathsf{cat}\,(\phi(v)) \in \mathit{Cats}$, $\mathbf{v}$ and $\phi(\mathbf{v})$ are sets of categorized nodes and $|\mathbf{v}|_c = |\phi(\mathbf{v})|_c$[3] for each category $c \in \mathit{Cats}$, and there exists a hyperedge $\phi(\mathbf{v}) \rightarrow \phi(\{v\}) \in E_{\mathcal{A}}{}'$.

In Definition 8, we define coarse homomorphic sequence diagrams by requiring (1) corresponding nodes be equivalent for each diagram, (2) for each corresponding hyperedge the

[3] $|S|$ refers to the cardinality of a set S.

number of origin nodes of each category are equivalent and the target nodes of the hyperedge is of the same category, and (3) each hyperedge has a corresponding hyperedge in both sequence diagrams.

Definition 9 (*Strict Sequence Diagram Homomorphism*) Let $\psi : H \to H'$ be a coarse sequence diagram homomorphism between code hypergraphs $H(N, E_{\mathcal{A}})$ and $H'(N', E_{\mathcal{A}}')$. Then, $\psi : H \to H'$ is a *strict sequence diagram homomorphism* if for $v \in N$ and $\mathbf{v} \in 2^V$ such that $(\mathbf{v} \to v) \in E_{\mathcal{A}}$, v and $\psi(v)$ are code object nodes in which $\mathsf{Datatype}(v) = \mathsf{Datatype}(\psi(v))$ and $\mathbf{v}$ and $\psi(\mathbf{v})$ are sets of code object nodes such that for all i, $\mathsf{Datatype}(\mathbf{v}_i) = \mathsf{Datatype}(\psi(\mathbf{v}_i))$.

Definition 9 defines strict homomorphism among sequence diagrams by insisting the type of an object and its image under ψ be the same type. We may now define a coarse and strict sequence diagram isomorphism based on the structural requirements of the coarse sequence diagram homomorphism.

Definition 10 (*Sequence Diagram Isomorphism*) ϕ is a *coarse sequence diagram isomorphism* if (i) ϕ is a bijection, (ii) ϕ is a coarse sequence diagram homomorphism, and (iii) ϕ^{-1} is a coarse sequence diagram homomorphism. If ϕ is a coarse sequence diagram isomorphism between H and H', we may write $H \simeq H'$. We similarly define a *strict sequence diagram isomorphism* based on strict sequence diagram homomorphism between H and H' and write $H \cong H'$.

Definition 11 (*Analogous Sequence Diagram*) Sequence diagrams $\mathcal{D}_1$ and $\mathcal{D}_2$ are *analogous* if $\mathcal{D}_1 \simeq \mathcal{D}_2$.

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 12 (*Depth of a Sequence Diagram*) The *depth* of a sequence diagram $\mathcal{D}$ 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 $1 \leq j \leq d-2$, $m_j \to m_{j+1}$ (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 $m_j \to m$ and $m \to m_{j+1}$.

As an example, the depth of the sequence diagram in Fig. 5 is depth 2; we observe a depth of 3 in the sequence diagram in Fig. 21 when excluding the user.

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 $\mathsf{Msg}(H) = |\mathsf{CallEdges}(E_{\mathcal{A}})|$ where $\mathsf{Msg} : H \to \mathbb{N}$. 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.

We define a binary relation $\sqsubseteq$ over the set of sequence diagrams $\mathcal{D}$ where for $\mathcal{D}_1, \mathcal{D}_2 \in \mathcal{D}$, $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$ if and only if $\mathsf{Msg}(\mathcal{D}_1) \leq \mathsf{Msg}(\mathcal{D}_2)$. $\sqsubseteq$ defines a partial order over the set of all sequence diagrams $\mathcal{D}$.

Theorem 1 *$\sqsubseteq$ defines a partial order over the set of all sequence diagrams $\mathcal{D}$.*

Proof $\sqsubseteq$ follows as a partial order over $\mathcal{D}$ since $\leq$ is a total order over the natural numbers. □

Before defining *interesting* for sequence diagrams, we define a function $\mathsf{top}(S, k)$ that selects the maximal k elements from a partially ordered set S (ties broken arbitrarily).

Let $\mathcal{D}_P$ be the set of all sequence diagrams for a program P. For $\mathcal{D} \subseteq \mathcal{D}_P$, let $Msgs(\mathcal{D}) = \{u \mid \exists D \in \mathcal{D} \text{ s.t. } \mathsf{Msg}(D) = u \in \mathbb{N}\}$ and let $Msgs(\mathcal{D})_k$ denote the multiset of the k greatest elements of $Msgs(\mathcal{D})$ where $1 \leq k \leq |\mathcal{D}|$. We define a function $\mathsf{select} : \mathbb{N} \to \mathcal{D}_P$ 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 13 (*Interesting Sequence Diagram*) For a program P with the set of sequence diagrams $\mathcal{D}_P$ with $|\mathcal{D}_P| = n$ and a fixed $0 < k \leq n$, $\mathcal{D}_P$ is an *interesting sequence diagram* if $\mathcal{D}_P \in \mathsf{top}(\mathcal{D}_P, k)$.

Consider generating a set of sequence diagrams $\mathcal{D}$ for the code in Fig. 2. $\mathcal{D}$ would include eight sequence diagrams from `middleButtonOnClick` (similar to Fig. 5); each of these diagrams has 6 messages. However, our generation routine may also construct Fig. 6 as an element in $\mathcal{D}$ corresponding to method `onOptionsItemSelected` in Fig. 2. Thus, $|\mathcal{D}| = 9$ with one diagram having 2 messages and eight diagrams having 6 messages. The only way for the sequence diagram in Fig. 6 to be considered interesting is if

Algorithm 1 Construction of a Code Hypergraph from a TCFG

```
1: procedure ConstructHypergraph(T: TCFG)
2:     Code-Hypergraph H;
3:     NodesH, T;
4:     HyperedgesH;
5:     return H
6: end procedure
```

$k = 9$; that is, the entire set of sequence diagrams $\mathcal{D}$ would be considered interesting. The sequence diagram in Fig. 6 is uninteresting for $1 \leq k \leq 8$ since it has the lowest message count compared to the other eight sequence diagrams. Hence, the sequence diagram in Fig. 5 will be considered interesting when $k = 8$ and, since ties are broken arbitrarily, may be considered interesting when $1 \leq k < 8$.

While it is arguable that Definition 13 may not be ideal for every user, we believe that code complexity is often rooted in the number of method invocations [9] and thus the probability is greater that a single trace can provide more information and thus is more likely to expose bugs and vulnerabilities.

Sequence diagrams can be used for debugging purposes, but can also provide a metric for code coverage in testing. We propose a definition that encompasses the notion of code coverage [6] using sequence diagrams.

Definition 14 (*Code Coverage*) Let m be a method and $H(N, E_{\mathcal{A}})$ be a code hypergraph. Also let $T \subseteq N$ be the set of all trace nodes for m. A hypergraph sequence diagram $\mathcal{D}$ containing a trace node $t \in T$ is said to *method cover* m with coverage rate $\frac{1}{|T|}$. For a decision instruction b in m with $|b|$ branches, a hypergraph sequence diagram $\mathcal{D}$ containing a trace node $t \in T$ is said to *branch cover* m with coverage rate $\frac{1}{|b|}$. A set of sequence diagrams provides *complete coverage* if the coverage rate is 100%.

Algorithm 2 Construction of Nodes for a Code Hypergraph

```
1: procedure Nodes(H: Code-Hypergraph, T: TCFG)
2:     n : TCFG-Node;
3:     for all n ∈ T do
4:         if n.isTypedNode() then
5:             H.AddNode(n);
6:         else if n.isMethod() then
7:             C ← T.ConstructTraces(n)
8:             for all t ∈ C do
9:                 H.AddNode(t);
10:            end for
11:        end if
12:    end for
13: end procedure
```

Algorithm 3 Construction of Hyperedges for a Code Hypergraph

```
1: procedure Hyperedges(H(N, E): Code-Hypergraph)
2:     N : Hypergraph-Node;
3:     m : String;
4:     c : Method-Call;
5:     r : Return-Type;
6:     for all N ∈ N where cat(N) = trace do
7:         for all c ∈ N do
8:             m ← c.methodName()
9:             C ← H.CollectTraceNodes(c, m)
10:            S ← {N} ∪ ⋃_{p∈c.Params} H.GetNode(p)
11:            for all t ∈ C do
12:                r ← t.getReturnObject()
13:                if r ≠ null then
14:                    n_r ← H.GetNode(r)
15:                    a ← m+ "Return";
16:                    H.AddHyperedge({t}, n_r, ⟨a, ∅⟩)
17:                else
18:                    H.AddHyperedge(S, t, ⟨m, ∅⟩)
19:                end if
20:            end for
21:        end for
22:    end for
23: end procedure
```

3 Constructing the hypergraph

In this section, we describe in more detail how *StaticGen* constructs a *code hypergraph* according to Algorithm 1. The basic input is a set of code files in an object-oriented language. We assume that the code is processed by an intermediate system [21,39] into a TCFG.

Nodes Given a TCFG $\mathcal{T}$, we construct the nodes of the *code hypergraph* H followed by the hyperedges. As in Definition 6, there are two types of nodes in a *code hypergraph*. With each node in a *code hypergraph* we associate one of two categories $Cats = \{\mathsf{object}, \mathsf{trace}\}$; we define a corresponding function cat that maps a node to its category. We may then define a function trace to extract the trace nodes from a set of nodes N, $\mathsf{trace}(N) = \{n \mid n \in N \wedge \mathsf{cat}(n) = \mathsf{trace}\}$.

In Algorithm 2, if a particular node $n \in \mathcal{T}$ is a declaration or a formal parameter, we add a corresponding code object node to H. The code hypergraph in Fig. 4 is defined with the code objects corresponding to formal parameters in Fig. 2: `v`, `b`, and `c`. Figure 4 includes illustrative code objects (`goodId`) declared in the corresponding code in Fig. 2 while omitting some others (`btnID`, `r`, and `id`).

If node n defines a method prototype for m, we construct all possible traces for m by a process that identifies all possible naive execution paths for m over $\mathcal{T}$ (on Line 7 using `ConstructTraces`). For each trace of method m, we add the corresponding node to H (Line 8 to Line 10). Hence, the complete code hypergraph corresponding to Fig. 2 would contain 8 trace nodes describing `middleButtonOnClick` and a single trace node describing the single paths for each method `get Red`, `getBlue`, `SetBtnColor`, and `SetUpperRightButton`.

Hyperedges We consider the two varieties of hyperedge 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. In Algorithm 3, we traverse H seeking the set of trace nodes N. For each trace node $N \in \mathcal{N}$ with method call c and method name m (Line 6 to Line 8), we construct a hyperedge e for H. The source nodes for e consist of the callee trace node t and the set of nodes corresponding to the actual parameters in the method call c (Line 10). The target node for e is a node corresponding to a trace of the called method. The annotation for e consists of the method name and a user-defined program state (default is empty) or propagated from the previous hyperedge.

In the source code of Fig. 2, we observe that `middle ButtonOnClick` calls several methods; most prominent in our example are `SetUpperRightButton` and `getBlue`. Thus, in the code hypergraph (Fig. 4), we observe two corresponding call hyperedges for these method invocations.

We note that the function $CollectTraceNodes$ on Line 9 selects all applicable trace nodes that are consistent with the class defining c and method m; that is, $CollectTraceNodes$ will return all trace nodes in the proper class and all of its super-classes, as applicable. Thus, due to restrictions of the capabilities of a front-end static analysis tool, it is possible that some trace nodes may not be collected by $CollectTraceNodes$; thus, possible polymorphic executions may be omitted from the code hypergraph.

For non-void methods (Line 12 to Line 16, Algorithm 3), we construct a return hyperedge r relating the current trace node t and the object being returned from t (using `getReturnObject` on Line 12) and adding to H. The annotation for r does not correspond to any existing function, so we append "Return" to the current method name m (to distinguish annotations accordingly) and a user-defined program state. Figure 4 contains the annotated return hyperedges `getBlueReturn` and `getIdReturn` since, respectively, `getBlue` and `getId` return code objects.

Lemma 3 *Every sequence diagram that describes a trace*[4] *of a program can be mapped bijectively to a hypergraph sequence diagram in the corresponding code hypergraph.*

Proof Let $H\,(N, E_{\mathcal{A}})$ be a code hypergraph corresponding to a program P. Let $m : \mathcal{D} \rightarrow HD$ be a mapping from the set of sequence diagrams $\mathcal{D}$ to the set of hypergraph sequence diagrams HD from H. Also let $d_1, d_2 \in \mathcal{D}$ be sequence diagrams such that $m\,(d_1) = m\,(d_2) \in HD$. In both cases, $m\,(d_1)$ and $m\,(d_2)$ are hyperpaths in H. Each such hyperpath is unique in H and corresponds to a trace in program P. It follows $d_1 = d_2$. Now, let $h \in HD$. Then, there exists a hyperpath p corresponding to h in H. Since a hyperpath corresponds to a trace t of program P, there exists a sequence diagram d that describes t. That is, $m\,(d) = h$. It follows m is bijective. □

Lemma 4 *Algorithm 1 constructs a code hypergraph containing all hypergraph sequence diagrams that correspond to sequence diagrams describing traces of a program P.*

Proof Let $H\,(N, E_{\mathcal{A}})$ be a code hypergraph corresponding to P. Line 7 in Algorithm 2 ensures that N contains all traces of each method in P. For each method call in P, Algorithm 3 constructs a corresponding call hyperedge and return hyperedge maintaining sequential ordering of the instructions in P by Lemma 1. Together, the nodes and hyperedges capture all such traces of program P as the set of all hyperpaths in H. It follows that H contains all such hypergraph sequence diagrams describing P. □

Remark 1 Based on Lemma 3, we have that any sequence diagram d that corresponds to a trace T of a program P gets bijectively mapped to a hypergraph sequence diagram in the corresponding code hypergraph H. Lemma 4 states that Algorithm 1 constructs a code hypergraph H that contains all such hypergraph sequence diagrams corresponding to sequence diagrams describing traces of P. Thus, any trace of a program P has a corresponding hypergraph sequence diagram in the code hypergraph constructed by Algorithm 1.

4 Refinement of code hypergraphs through pebbling

In this section, we describe an important process for sub-hypergraph identification using a process called pebbling (Sect. 4.1) as well as the impact of this procedure on code hypergraphs for generation of sequence diagrams (Sect. 4.2).

4.1 Sub-hypergraph identification through pebbling

As stated in Definition 2, each hyperedge is annotated with a parameterized set of values $A \in \mathcal{A}$ defined in the generation space. The *code hypergraph* of Definition 6 annotates each hyperedge with method name and program state information. The *code hypergraph* constructed in Sect. 3 is a detailed structure representing a set of input code. A naive generation of sequence diagrams would produce an exorbitant number of diagrams, possibly of great depth, length, and detail. For this reason, we must have a method to either include or exclude a set of nodes or a set of hyperedges; this inclusion process we call pebbling. We more formally define this notion as a pebbled hypergraph in Definition 15 as computed using the *pebbling* process specified in Algorithm 4.

Definition 15 (*Pebbled Code Hypergraph*) Let $H\,(N, E_{\mathcal{A}})$ be a hypergraph with $N_P \subseteq N$ a subset of hypernodes and

[4] A trace may or may not be executable, i.e., it is possible that a particular trace may never be executed.

Algorithm 4 Sub-Hypergraph Identification Through Pebbling

```
1: procedure PEBBLE(Hypergraph H (N, E_A), N_P ⊆ N, A_P ⊆ A)
2:     Hypergraph P;                                  ▷ Pebbled Sub-Hypergraph
3:     Worklist W ← N_P;
4:     while !W.empty() do
5:         n ← W.dequeue()                            ▷ Acquire a node
6:         if n.pebbled() then continue
7:         n.pebble();                                ▷ Mark the node
8:         P.AddNode(n);
9:         for all e ∈ n.edges do
10:                                  ▷ Consider allowable hyperedges
11:            if e.annotation.consistentWith(A_P) then
12:                                  ▷ If all origin nodes are pebbled,
13:                                  ▷ propagate target forward
14:                if e.pebbled() and !e.target.pebbled() then
15:                    P.AddHyperedge(e);
16:                    W.enqueue(e.target);
17:                end if
18:            end if
19:        end for
20:    end while
21:    return P;
22: end procedure
```

a subset of annotations $\mathcal{A}_P \subseteq \mathcal{A}$. Then, $H_P\left(N_P, E_{\mathcal{A}_P}\right)$ is a *pebbled annotated hypergraph* containing only reachable nodes and hyperedges consistent with $\mathcal{A}_P$: N_P and $E_{\mathcal{A}_P}$, respectively. Similarly, a *pebbled code hypergraph* consists of the reachable nodes and hyperedges from a code hypergraph.

Algorithm 4 is a modification of the classic marking algorithm as first defined by Dowling and Gallier [13] for satisfiability of propositional horn clauses. Pebbling is a linear-time traversal over an annotated hypergraph that identifies the sub-hypergraph [7] consistent with user specified information. For example, if the user disallows library methods, those hyperedges with annotations containing `System.` or `toString` in Java, would not be allowed in the resulting pebbled code hypergraph. As described in Algorithm 4, pebbling is a breadth-first traversal over an annotated hypergraph where we mark each node with a pebble once it is visited (Line 7). Then, on Line 9 through Line 19, we use the following rule for pebbling and propagation: if all origin nodes of a hyperedge are pebbled, we place the target node of the hyperedge in the work list. As pebbling continues, we add all "unexpanded," pebbled nodes (Line 8) and hyperedges (Line 15) to the sub-hypergraph in preparation for the return of the pebbled version of the hypergraph (Line 21).

For example, in Fig. 4, 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`.

Lemma 5 (Correctness of Algorithm 4) *Given a code hypergraph $H\,(N, E_{\mathcal{A}})$, Algorithm 4 returns a pebbled code hypergraph $H_P\left(N_P, E_{\mathcal{A}_P}\right)$.*

Proof Given a code hypergraph $H\,(N, E_{\mathcal{A}})$ and a set of input nodes N_P, Algorithm 4 constructs a new code hypergraph P. The loop from Line 4 to Line 20 requires the worklist queue W to be empty. Each node that is added to W (Line 3 and Line 16) is dequeued as node n on Line 5 and added to P. The loop beginning on Line 9 then considers each hyperedge $e = (S, t, a) \in E_{\mathcal{A}}$ for which $n \in S$. If all S are pebbled on Line 7, t is reachable and thus e is added to P and t is added to W (and eventually added to P). P thus contains N_P and all reachable nodes from N_P via the set of fully pebbled hyperedges in P. □

4.2 Pebbling a code hypergraph

Our generation framework for sequence diagrams is based on the concept of "objects as messages." That is, a method call (message) will only be invoked when all of the constituent actual parameters (objects) are in one-to-one correspondence with the formal parameters are available to initiate the message; that is, all origin nodes of a hyperedge have been pebbled. The pebbling procedure described in Sect. 4.1 is a general formalization of this notion, but with respect to sequence diagrams, pebbling is a selection process where only messages that are ready and able to be communicated appear in the resultant sequence diagrams. Branches in the original code may result in an object not being initialized, or in general, an object is not ready for communication and thus will not be pebbled.

Recursion and Pebbling We have mentioned a limitation in our approach how intra-method loops are treated as conditional expressions. However, we must also consider loop via method calls: recursion. It is easy to conceive of an example with three objects O_1, O_2, and O_3 engage in the following method calls:

$$main() \rightarrow O_1.a() \rightarrow O_2.b() \rightarrow O_3.c() \rightarrow O_1.a().$$

While the code hypergraph constructed from the TCFG would encode this recursive instruction sequence as a cycle, the resulting pebbled code hypergraph will be acyclic. We refer to Algorithm 4 and argue informally that each node in a code hypergraph will only be expanded once. Pebbling is a worklist-based procedure that will add a node n to the list for processing when it is "discovered" on Line 3 or Line 16. When n is taken from the list on Line 5, it is first marked (pebbled) on Line 7 and then "expanded." If n is ever added to the worklist again as being reachable via some hyperedge on Line 16, the conditional on Line 6 will prohibit expansion of n and inclusion of the hyperedge in the resulting pebbled

Algorithm 5 Sequential Hyperpath Collection Construction for a Method m

```
1: procedure PATHS(H: Code-Hypergraph, m: Method, O: Objects)
2:     Y ← ∅;
3:     for all C ∈ H.CollectTraceNodes(m) do
4:         Y ← Y ∪ PATHSFROMTRACE H, C, O;
5:     end for
6:     return Y;
7: end procedure
```

Algorithm 6 Trace Node-Based Hyperpath Construction

```
1: procedure PATHSFROMTRACE(H: Code-Hypergraph, T: trace, O: Objects)
2:     Y ← {∅};                                   ▷ Set of Paths
3:     for all i ∈ T do
4:         if i.isInvocation() then
5:             e ← H.getHyperedge(i);
6:             A ← O ∪ e.typeNodes;
7:             Q ← PATHS H, e.method(), A
8:         else
9:             Q ← {i};
10:        end if
11:        Y ← APPENDPATHS Y, Q;
12:    end for
13:    return Y;
14: end procedure
15: procedure APPENDPATHS(Y: Path Set, Q: Path Set, A: Objects)
16:    U ← ∅;                                     ▷ Path Accumulator
17:    for all Y ∈ Y do
18:        for all q ∈ Q do                       ▷ Append Paths and Objects
19:            Y.Path ← Y.Path + q.Path;
20:            Y.O ← Y.O ∪ A ∪ q.O;
21:            U ← U ∪ { Y };
22:        end for
23:    end for
24:    return U;
25: end procedure
```

code hypergraph. The result of this breadth-first procedure is an acyclic code hypergraph.

For generation of sequence diagrams, we assume any code hypergraph has been pebbled according to Algorithm 4 resulting in a *pebbled code hypergraph*.

5 Static sequence diagram construction

Sequence diagram generation consists of three phases: (1) sub-hypergraph identification through pebbling [13] as described in Sect. 4, (2) hyperpath identification, and (3) converting from a hyperpath to a sequence diagram.

5.1 Hyperpath identification

For a given method m, we can construct the corresponding set of all paths in a *code hypergraph* H using Algorithm 5 (which relies upon Algorithm 6). We note that this construction algorithm maintains the same information for a sequence diagram as stated in Definition 7, but instead of maintaining a sequential hyperpath the result is an equivalent path consisting of one-to-one edges and an associated set of objects. A method has many associated traces so PATHSH, m in Algorithm 5 acquires all such traces and accumulates the set of all paths. For each trace (acquired using `CollectTraceNodes` on Line 3), we construct all paths. In this case, a path is a DAG whose nodes are code instructions and edges are attributed to the flow of control of the code. For a path $\mathcal{P}$, we use an object-oriented notation in which (1) $\mathcal{P}.Path$ refers to the path associated with a trace path and (2) $\mathcal{P}.\mathcal{O}$ is an associated set of code objects.

Recall that a trace T is a sequential set of instructions. In Algorithm 6, PATHSFROMTRACEH, T loops starting on Line 3 through the set of instructions in T using i as our iterator representing an instruction seeking method calls. The Boolean function `isInvocation` on Line 4 will return true if the instruction is a method call. If the instruction in T is a method call, hyperpath construction is recursive (Line 7) by following the corresponding call hyperedge to the target trace node, otherwise, we simply view the instruction i as a singleton set (Line 9). Regardless of how instruction i is processed, we append the new sets of paths ($\mathcal{Q}$) to the old path sets ($\mathcal{Y}$) using procedure APPENDPATHS$\mathcal{Y}$, $\mathcal{Q}$; this includes the objects in a particular path (Line 20). We assume set union with $\cup$ operator maintains sequential ordering of instructions and creates a directed edge from the last instruction of one path to the first instruction of the other path.

Lemma 6 (Correctness of Algorithm 5) *For a code hypergraph H and a method m, Algorithm 5 collects all hyperpaths for m.*

Proof We consider a single trace node C for a method m as it is clear that the loop beginning on Line 3 of Algorithm 5 collects all hyperpaths corresponding to traces of method m. Algorithm 6 considers each instruction in C. In the base case, if C contains no method calls, $\mathcal{Y}$ will contain a single hyperpath consisting of a single node with no edges. In the inductive case, if C contains a method call to a method μ, we append all recursively constructed hyperpaths with μ as root (Line 7) to each element of $\mathcal{Y}$ (Line 11). Each such method invocation in C is expanded and appended similarly. Algorithm 5 thus exhaustively constructs all hyperpaths for m using a recursive, tree-style formulation. □

5.2 Generating sequence diagrams from a hyperpath

As described in Sect. 5.1, Algorithm 5 computes the set of all paths corresponding to sequential hyperpaths, including the associated objects with the path. Given a path, we construct the associated sequence diagram using Algorithm 7. The set of objects in the sequence diagram corresponds to

Algorithm 7 Converting a Path to a Sequence Diagram

```
1: procedure PathToDiagram(P: Path)
2:     D: Sequence Diagram;
3:     D.O ← P.O;
4:     for all i ∈ P do
5:         if i.isInvocation() then
6:             D.addMessage(P.getEdge(i));
7:             q ← PathToDiagramP.getPath(i);
8:             D ← D ∪ q;
9:             D.addReturnMessage();
10:        end if
11:    end for
12:    return D;
13: end procedure
```

the set of objects in the path (Line 3). Messages in the sequence diagram are constructed recursively; each method invocation instruction results in a sub-sequence diagram, q being constructed (Line 7) and appended to the main sequence diagram, D (Line 8), including a return message using `addReturnMessage`.

We consider the code hypergraph in Fig. 4 which depicts a single hyperpath corresponding to the code in Fig. 2. In this example, the consequent code objects in the hyperpath is an anonymous object of type `Main` as well as a code object of type `Button` (`b`); no methods act on code object `c` thus it does not appear in the resulting sequence diagram. To generate the set of method calls for the sequence diagram, we begin with the dashed hyperedge into the trace node for `middleButtonOnClick`. Considering each of the instructions of this trace node in turn, we add messages in causal order: recursively following call hyperedges to `getBlue` and `SetUpperRightButton` then subsequently following their respective return hyperedges. The result is the sequence diagram in Fig. 5.

6 Interface for diagram generation

Statically generating sequence diagrams is a useful tool for a programmer to perform software archeology. Whether the user is examining a legacy system or their own code, the user is a critical element in successful use of *StaticGen*. In this section, we provide a user's perspective to interacting with our system to obtain a desired set of sequence diagrams.

A sequence diagram D has features such as: depth as defined in Definition 12, 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.

6.1 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 $j \geq 1$.

Lemma 7 *A hyperpath Y in a* code hypergraph *H is a string in $\mathcal{L}$.*

Proof It is clear that a topological sort of the graph corresponding to Y results in a unique 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. 5; 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 $\mathcal{D}_S$, 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 $(\mathcal{D}, k)$.

- "`method cover` p $\mathcal{D}$" and "`branch cover` p $\mathcal{D}$" each return sequence diagrams ensuring minimal method and branch coverage, respectively, for a lower bound percentage p.

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.

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

The code hypergraph is a representation of all traces of a program; it is thus a prohibitively large structure. We therefore give the user the power to define queries that will make the analysis tractable and the resulting set of sequence diagrams more likely to be meaningful. For example, a user may not want to dive into a complete analysis of a program from a `main` method, instead they may begin their analysis at any arbitrary method. Our solution is a set of *modes of refinement*. A user-defined query is a tuple of *refinement mode*s or *global query predicates*. A *refinement mode* is of the form $\langle t, m \rangle$ where t is a unique identifier for an object or a method and m is a mode descriptor; we define each of the mode descriptors below.

- Synthesis can be performed on-demand beginning at any method using "`start` M," where M is a method from which the resulting diagram(s) will begin.
- Since source code may contain method calls to library-based functionality, "`filter` S" where S is a set of methods or code objects, removes messages or objects in a sequence diagram corresponding to all $s \in S$.
- Synthesis may result in a set of sequence diagrams. For a set of predicates R, "`remove` R" will remove all resulting sequence diagrams for which all $r \in R$ evaluate to `true`.
- For a set of predicates R, "`accept` R" collects all sequence diagrams for which all $r \in R$ evaluate to `true`.

Beyond refinement modes, global query predicates prune the resulting set of sequence diagrams. Using our definition for interesting sequence diagrams, a user may use the predicate "`top-interesting` d" where d is an integer specifying the number of resulting diagrams. Similarly for code coverage, we may refer to "`function-cover` p" or "`branch cover` p" where p is a lower bound percentage of the coverage desired.

6.2 Query language semantics

In this subsection, we formalize the query language semantics by describing the denotational interpretation of the query language and operations defined in Sect. 6.1.

We define D to be the set of string objects over the language of sequence diagram strings $\mathcal{L}$. These sequence diagram string objects also maintain diagram information including, but not limited to the number of messages, branch coverage, and method coverage. Our query language consists of a set of predicates, R, over a set of set of sequence diagrams, D. We use emphatic brackets $[\![\ ]\!]$ to express denotations of expressions in the query language. We first consider the set of operations related to evaluating predicates. For the set of predicates R, we evaluate a sequence diagram $\mathsf{d} \in \mathsf{D}$ to confirm if it adheres to a subset of predicates in R: we define $\mathsf{Eval} : (D \times 2^R) \rightarrow \{true, false\}$. We then define a function E which evaluates a given operation BoolOp, $\mathsf{E} : \mathsf{BoolOp} \rightarrow \mathsf{Eval}$. Since our operations result in boolean expressions, we define standard notions of logic-based commands, BoolOp. For $\varepsilon, \varepsilon' \in \mathsf{Eval}$, $\mathsf{d} \in \mathsf{D}$, and predicates $R' \in 2^R$:

- $\mathsf{E}[\![\ \varepsilon\ \textbf{\textit{and}}\ \varepsilon'\ ]\!]\ (\mathsf{d}, R') = \mathsf{E}[\![\varepsilon]\!]\ (\mathsf{d}, R')\ \wedge\ \mathsf{E}[\![\varepsilon']\!]\ (\mathsf{d}, R')$,
- $\mathsf{E}[\![\ \varepsilon\ \textbf{\textit{or}}\ \varepsilon'\ ]\!]\ (\mathsf{d}, R') = \mathsf{E}[\![\varepsilon]\!]\ (\mathsf{d}, R')\ \vee\ \mathsf{E}[\![\varepsilon']\!]\ (\mathsf{d}, R')$,
- $\mathsf{E}[\![\ \textbf{\textit{not}}\ \varepsilon\ ]\!]\ (\mathsf{d}, R') = \neg \mathsf{E}[\![\varepsilon]\!]\ (\mathsf{d}, R')$,
- $\mathsf{E}[\![\ \textbf{\textit{true}}\ ]\!]\ (\mathsf{d}, R') = true$,
- $\mathsf{E}[\![\ \textbf{\textit{false}}\ ]\!]\ (\mathsf{d}, R') = false$, and
- $\mathsf{E}[\![\ \mathsf{d}\ ]\!]\ (\mathsf{d}, \{r\} \subseteq R') \in \{true, false\}$.

The last expression is the boolean evaluation of a single predicate over a single sequence diagram.

For language operation commands Oper, we define a function $\mathsf{C} : \mathsf{Oper} \rightarrow ((2^{\mathsf{D}} \times 2^R) \rightarrow 2^{\mathsf{D}})$, ensuring each query operation takes a set of sequence diagrams and returns a refined subset of sequence diagrams. We define sequential execution of operations $\mathsf{C}[\![\gamma_1; \gamma_2]\!]\ (\mathsf{d}, R') = \mathsf{C}[\![\gamma_1]\!]\ (\mathsf{d}, R') \circ \mathsf{C}[\![\gamma_2]\!]\ (\mathsf{d}, R')$ with $\circ$ referring to function composition. We then define a basic conditional operation: with $\varepsilon \in \mathsf{Eval}$ and $\gamma \in 2^{\mathsf{D}}$, $\gamma' ::= \textbf{\textit{if}}\ \varepsilon\ \textbf{\textit{then}}\ \gamma$. This expression is a traditional *if-then* statement where the missing "else" evaluates to an empty set:

$$\mathsf{C}[\![\ \textbf{\textit{if}}\ \varepsilon\ \textbf{\textit{then}}\ \gamma\ ]\!]\ (\{\mathsf{d}\}, R') = \mathsf{C}[\![\gamma]\!]\ (\{\mathsf{d}\}, R')$$

if $\mathsf{E}[\![\varepsilon]\!]\ (\{\mathsf{d}\}, R') = true$, otherwise $\emptyset$.

Before we define the denotational semantics of the query operations, we define two support operations. In some cases, the user may wish to exclude elements of a sequence diagram

such as a particular variable or method; the first support operation defines a means of such pruning mechanisms. Mutate is a function that maps a mutation operation, MutOp to a function that takes a sequence diagram and returns a modified version of the sequence diagram: Mutate : MutOp $\rightarrow$ (D $\rightarrow$ D). The second support function Property is a mapping from a property operation PropOp that takes a sequence diagram and returns a quantity related to the diagram (e.g., method count, method coverage, etc.): Property : PropOp $\rightarrow$ (D $\rightarrow$ $\mathbb{R}$).

We may now define the denotational semantics of the query operations.

To properly define filtration, we first define two mutation operations, one for code objects and one for methods.

For a string s and two characters a, z, we assume the standard substring operation $\mathsf{substr}(s, a, z)$ returns the string beginning at a through all characters ending with (and including) z. In the case where characters a or z is not found in s, $\mathsf{substr}(s, a, z)$ returns an empty string. We define a complementary operation $\mathsf{rem\_substr}(s, a, z)$ which returns the original string excluding all such substrings $\mathsf{substr}(s, a, z)$. If $\mathsf{substr}(s, a, z)$ does not appear in s, $\mathsf{rem\_substr}(s, a, z) = s$ by returning the original string, unmodified; $\mathsf{rem\_substr}(s, a, a) = a$ deletes all occurrences of character a from string s.

For a sequence diagram string d, a character $m.a$ corresponding to method m, and a character $m.z$ corresponding to the return of method m, $\texttt{prune}(\mathcal{D}, m) = \mathsf{rem\_substr}(\mathcal{D}, m.a, m.z) \in \mathsf{MutOp}$ removes method m from d.

Pruning a code object c from a sequence diagram is a more involved process since it requires we prune c as well as all methods that act on c. For a sequence diagram string d and a character o corresponding to code object c, we prune all methods m_i $0 < i \leq k$ that call a method defined by code object c; we define functionally with a functional-style pattern matching and list dissection.

$$
\begin{aligned}
&\texttt{prune}(\mathsf{d}, o) &&= \mathsf{rem\_substr}(\mathsf{d}, o, o)\\
&\texttt{prune}(\mathsf{d}, m) &&= \mathsf{rem\_substr}(\mathsf{d}, m.a, m.z)\\
&\texttt{prune}(\mathsf{d}, mH :: mT) &&= \texttt{prune}(\texttt{prune}(\mathsf{d}, mH), mT)\\
&\texttt{prune}(\mathsf{d}, o) &&= \texttt{prune}(\texttt{prune}(\mathsf{d}, o.m_i), o, o)
\end{aligned}
$$

The first function removes the actual code object c from the diagram (invoked last). The second function removes all occurrences of method m from the sequence diagram. The third function iterates through all methods for which code object c interacts. The final function invokes all method removal and code object removal, respectively.

With prune defined, we may now filter a single diagram with a set of predicates: $\gamma :: = \texttt{filter}(\{\mathsf{d}\}, L) = \texttt{prune}(\mathsf{d}, L)$. Filtering many diagrams over a set of predicates: $\gamma :: = \texttt{filter}(\mathsf{D}, L) = \bigcup_{\mathsf{d} \in \mathsf{D}} \texttt{filter}(\{\mathsf{d}\}, \ell)$.

For clarity and simplicity, we define an evaluator to determine if a diagram d satisfies all predicates in a set R.

$$b :: = \texttt{satisfies}(R, \mathsf{d}) = \bigwedge_{r \in R} \mathsf{E}[\![\mathsf{d}]\!](\{\mathsf{d}\}, \{r\})$$

For a set of predicates R and a sequence diagram d, a diagram is *removed* if $\texttt{remove}(R, \mathsf{d})$ evaluates to true; that is, diagram d does not satisfy all $r \in R$:
$\gamma :: = \texttt{remove}(R, \mathsf{d}) =$
$\mathsf{C}[\![\ \textbf{\textit{if}}\ \ \textbf{\textit{not}}\ (\texttt{satisfies}(R, \mathsf{d}))\ \textbf{\textit{then}}\ \{\mathsf{d}\}\ ]\!](\{\mathsf{d}\}, R')$.
Similarly, for a set of diagrams D, $\texttt{remove}(R, \mathsf{D}) = \bigcup_{\mathsf{d} \in \mathsf{D}} \texttt{remove}(R, \{\mathsf{d}\})$.

Similarly, $\texttt{accept}(R, \mathsf{d})$ requires d satisfy all $r \in R$:

$$
\begin{aligned}
\gamma :: &= \texttt{accept}(R, \mathsf{d})\\
&= \mathsf{C}[\![\ \textbf{\textit{if}}\ \texttt{satisfies}(R, \mathsf{d})\ \textbf{\textit{then}}\ \{\mathsf{d}\}\ ]\!](\{\mathsf{d}\}, R').
\end{aligned}
$$

For a set of sequence diagrams D, $\texttt{accept}(R, \mathsf{D}) = \bigcup_{\mathsf{d} \in \mathsf{D}} \texttt{accept}(R, \mathsf{d})$. We note that accept and remove have similar, yet opposing goals where an accept command followed by a remove with the same set of predicates results in an empty set:

$$\texttt{remove}\,(R, \texttt{accept}(R, \mathsf{D})) = \emptyset.$$

This observation is also true when remove precedes accept: $\texttt{accept}\,(R, \texttt{remove}(R, \mathsf{D})) = \emptyset$.

Let $\mathsf{D} = \{\mathsf{d}_i\}$ be a set of sequence diagrams ordered by the number of messages: for all $1 \leq i < j \leq |\mathsf{D}|$, $\mathsf{Msg}\,(\mathsf{d}_i) \leq \mathsf{Msg}\,(\mathsf{d}_j)$. In [11], the authors define the function $\texttt{select}(k, S)$ which returns the k-th largest element in a set S. We modify this notion to define $\texttt{select}(k, \mathsf{D})$ to return the sequence diagram corresponding to the k-th largest number of messages of the sequence diagrams in D. We select the top-k sequence diagrams:

$$
\begin{aligned}
\texttt{top}(\mathsf{D}, k) = \texttt{select}(k, \mathsf{D}) \cup\\
[\text{if } k + 1 <= |\mathsf{D}| \text{ then } \texttt{top}(\mathsf{D}, k + 1)]
\end{aligned}
$$

Hence, $\gamma :: = \texttt{top-interesting}(k, \mathsf{D}) = \texttt{top}(\mathsf{D}, k)$ results in the set corresponding to D restricted to the top-k: $\{\mathsf{d} \in \mathsf{D} \mid \mathsf{d}_{|\mathsf{D}|-k+1}, \mathsf{d}_{|\mathsf{D}|-k+2}, \ldots, \mathsf{d}_{|\mathsf{D}|}\}$.

Sequence diagram object representations maintain code coverage metrics. For branch and method coverage, respectively,

$$
\begin{aligned}
&\texttt{branch cover}(p, d) =\\
&\quad \mathsf{C}[\![\ \textbf{\textit{if}}\ \mathsf{d}\ \textbf{\textit{then}}\ \{d\}\ ]\!](\mathsf{d}, \{\mathsf{d} \in \mathsf{D} \mid \texttt{branch}(\mathsf{d}) > p\})
\end{aligned}
$$

and

$$
\begin{aligned}
&\texttt{method-cover}(p, d) =\\
&\quad \mathsf{C}[\![\ \textbf{\textit{if}}\ \mathsf{d}\ \textbf{\textit{then}}\ \{d\}\ ]\!](\mathsf{d}, \{\mathsf{d} \in \mathsf{D} \mid \texttt{method}(\mathsf{d}) > p\}).
\end{aligned}
$$

Theorem 2 (Soundness and Completeness of Query Language Semantics) *For a set of sequence diagrams* D *over the language of sequence diagram strings* $\mathcal{L}$ *and a set of predicates* R.

(A) *For* $\mathsf{d} \in \mathsf{D}$ *and method* m, `filter` *results in a substring* d' *of* d *omitting* m.
(B) *For* $\mathsf{d} \in \mathsf{D}$ *and code object* o, `filter` *results in a substring* d' *of* d *omitting all dependencies of* o.
(C) *For* $\mathsf{D}' \subseteq \mathsf{D}$ *satisfying all predicates in* R, `remove` *results in the set* $\mathsf{D} \setminus \mathsf{D}'$.
(D) *For* $\mathsf{D}' \subseteq \mathsf{D}$ *satisfying all predicates in* R, `accept` *results in the set* D'.
(E) *For* $0 < k \leq |\mathsf{D}|$, `top-interesting` *returns* $\mathsf{D}' \subseteq \mathsf{D}$ *with* $|\mathsf{D}'| = k$, *the set of* k *sequence diagrams with the greatest number of messages.*
(F) `branch coverage` *(resp.* `method coverage`*) for* p *returns the subset* $\mathsf{D}' \subseteq \mathsf{D}$ *of sequence diagrams with coverage greater than* p.

Proof (C), (D), (E), (F) are clear by their definitions.

For (A), `filter`(d, m) prunes all occurrences of substrings from the character corresponding to the invocation of method m to the character corresponding to the return from m from the string corresponding to d. Filtering of a code object o in (B) prunes all methods in o and all methods strictly called by o as well as the character corresponding to o. □

6.3 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 libitum.

Within the bounds of the user selected query, we prioritize what the user sees by first eliminating strictly isomorphic diagrams 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 using the following greedy algorithm.

```
accept:method:Main.SetUpperRightButton(int)
accept:method:Main.SetUpperLeftButton(int)
```

Fig. 7 Example accept query for Fig. 2

1. Select the most interesting sequence diagram $s \in S$ that adds the most new information (number of new methods covered).
2. Remove s from S ($S := S \setminus \{s\}$).
3. Add s to I ($I := I \cup \{s\}$).
4. Repeat steps (1) through (3) until the desired coverage rate is acquired.

Using this construction, I is a minimal coverage set for the input code and provides a desired viewing order with the most unique information possible in the fewest views of diagrams, assuming a quality user query.

6.4 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, and one of which is shown in Fig. 5. If we append to Q an `accept` predicate with `SetUpperRightButton(int)`, the only diagram returned is shown in Fig. 5. As another example, Fig. 6 arises from a query requesting the least interesting diagram from analyzing all methods.

Queries can be used to select a more sophisticated set of diagrams. Queries can choose to accept diagrams with certain characteristics, for example the query in Fig. 7 will reduce the diagrams generated by the code in Fig. 2 to only the diagrams containing the methods `SetUpperRightButton(int)` or `SetUpperLeftButton(int)`. In a larger example, the complete package name of the method to be queried would be required. Figure 8 will take the set of all possible diagrams, and remove any diagram that contains the method `SetUpperRightButton(int)`. In the remaining diagrams, any calls to the method `nextInt(int)` will be omitted from the diagram. This will also necessarily remove any calls that `nextInt(int)` would make to any other methods. The diagrams would appear as if the call to `nextInt(int)` simply did not exist.

We also have the capacity to produce more sophisticated queries by combining accept or reject criteria with the or and

```
reject:method:Main.SetUpperRightButton(int)
filter:method:Random.nextInt(int)
```

Fig. 8 Example reject and filter query for Fig. 2

```
accept:method:Main.SetUpperLeftButton(int) &&
accept:method:Main.getRed() &&
reject:method:Main.getBlue()
```

Fig. 9 Example compound query for Fig. 2

and operations. In Fig. 9, we can see three methods that are combined with the and operator. In order for a diagram to be a member of the output set of this query, it must pass each of the components of that query. Therefore, it must contain both the methods `SetUpperLeftButton(int)` and `getRed()` and not have a call to the method `getBlue()`. This query language allows the user to select a very specific subset of the diagrams produced by the system. Additionally, the query framework produces an output both as diagrams and as a dataset of json files which can be used as another input to the querying system, allowing that subset to serve as a starting point for another round of querying.

6.5 Possible incompleteness of query language features

The query language defined in this section is based on the intuitive constructs that a software engineer may normally be looking for while trying to understand a program. For example, a software engineer may wish to use the tool for the following use cases: (1) they may filter a set of sequence diagrams based on some criteria, (2) remove sequence diagrams that they may not be interested in, (3) request a set of sequence diagrams that satisfy certain criteria, and (4) want to see the most interesting sequence diagrams that would lead to better understanding of the code. The query language attempts to formalize the intuitive notions of what a user requires. However, we admit that, over time, the features for which we have defined our query language should evolve to meet the needs of future users.

7 Experimental analysis

In this section, we describe several analyses we used to evaluate the effectiveness of our tool, *StaticGen*. We begin by describing our benchmark code (Sect. 7.1) as a basis for our timing (Sect. 7.2) and effectiveness (Sect. 7.3) of the tool. In Sect. 7.4, we then consider how *StaticGen* visualizes polymorphically defined methods and their invocations. Last, we compare *StaticGen* to an existing tool (Sect. 7.5).

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 3.2GHz with 8 GB RAM on 64-bit Linux Mint operating system.

7.1 Benchmark code

Our initial tests have focused on open-source Android bytecode applications taken from [12,44] with wide-ranging focus, including: ad blocking, email, and web browsing. The bytecode was input into the Soot framework [39] which can process bytecode or source code thus bringing the same capabilities to bear, independent of input format. Table 1 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 are a useful measurement of the complexity of the program analyzed. The operation count for our corpus is shown in Table 1.

As another measure of complexity of our corpus, the target Android code, we consider the histogram in Fig. 10 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. 10; the mean 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 11 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 $r^2 = 0.5643$.

7.2 Time and scope of synthesis

We measure tool efficiency by considering generation time. Our reported execution times include Soot's Simplification [39] procedure, hypergraph construction, diagram genera-

Table 1 Android application corpus

Project name	Package	Class #	Time (s)	Op #
AdBlockPlus	org.adblockplus. android	90	130.2	3481
APG	org.thialfihar. android.apg.ui	16	396.4	863
ConnectBot	org.connectbot	184	154.9	12102
CSipSimple	com.csipsimple. service	78	593.2	3736
Fennec	com.squareup. picasso	46	7.3	1468
Jitsi	org.jitsi.service	110	10.2	2835
K-9 Mail	com.fsck.k9. service	26	22.5	1595
Linphone	org.linphone.core	52	23.8	2192
Orbot	a.a.a.a	9	3.0	437
sipdroid	org.sipdroid.net	6	4.4	641
AcDisplay	com.achep. acdisplay.services	59	12.4	2588
AC Stopwatch	com.achep. stopwatch	184	628.4	11426
Active Notify	com.aky. peek.notification	119	69.2	6543
AdAway	org.adaway	126	105.8	5386
AppOps	com.ssrij.appops	15	2.09	116
Blackberry Unlocker	ir.irtci	12	3.25	268
Better Battery Stats	com.asksven. betterbatterystats	151	418.6	11612
Better Wifi On/Off	com.asksven. betterwifionoff. data	10	3.0	310
Color Clock	com.brianco. colorclock	17	2.9	316
Amaze File Manager	com.amaze. filemanager. adapters	35	23.4	2730
Complete Linux	com.zpwebsites. linuxonandroid	226	29.8	7407
CPU Spy Plus	com.cpuspy	7	2.98	186
Desk Clock	com	98	114.4	4965
Fifteen Puzzle	com	37	8.4	2089
Fontster	com.chromium. fontinstaller	112	112.1	4910
Halo Shortcuts	com	23	3.9	450
Heads Up	com.achep. headsup	20	9.2	1885
Jelly Bean Clock	com	16	2.7	254
Fake GPS Path	com.rc	33	3.1	783
Root Verifier	com	11	1.7	72

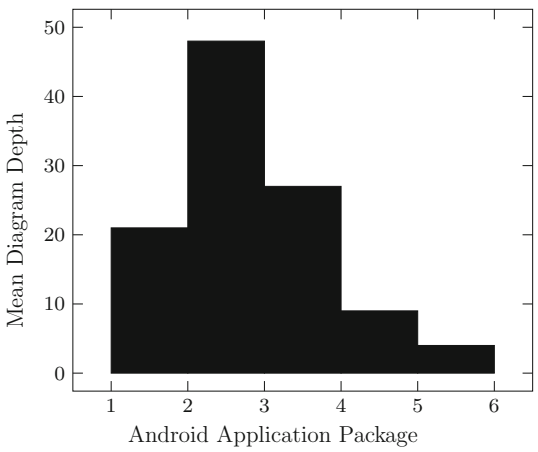

Fig. 10 Mean sequence diagram depth per android application package

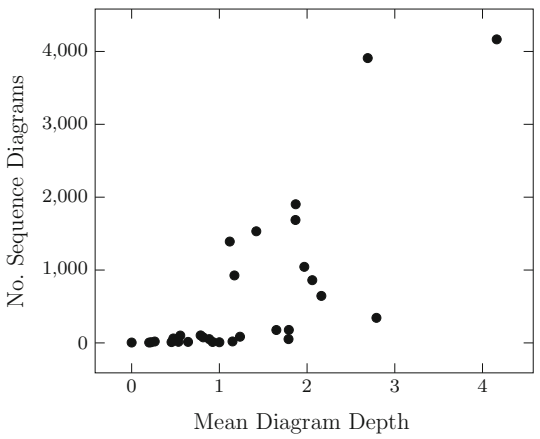

Fig. 11 No. generated sequence diagrams versus mean diagram depth for the entire corpus

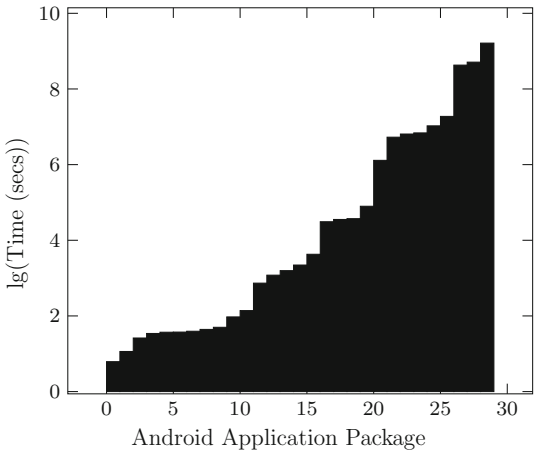

Fig. 12 Time per android application package

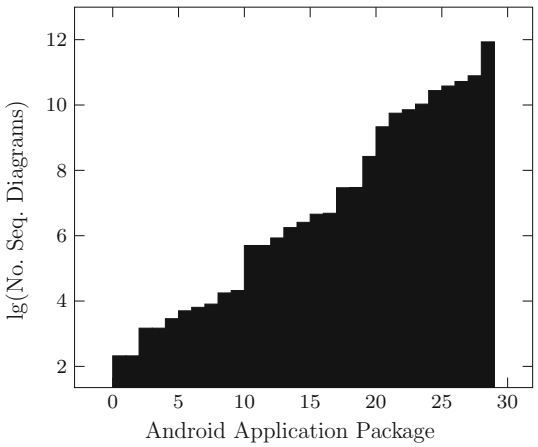

Fig. 13 Number of sequence diagrams per android application package

tion, and refinement. In Fig. 12, 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. 13 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.

7.3 Evaluation of interestingness and filtering

We test the usefulness of our interestingness metric by examining the possibility of using it in uncovering *security vulnerabilities in code* in concert with diagram filtering. We selected an independently studied example for assessing our definition of interestingness. Both Livshits [26] and Sampaio

```
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. 14 BlueBlog [8] doGet function

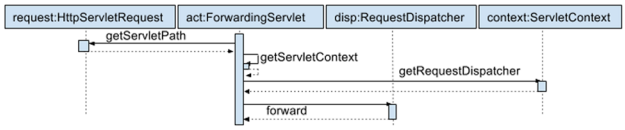

Fig. 15 BlueBlog [8] doGet sequence diagram fragment

[40] used a web application named blueblog [8] in their corpora of applications with security vulnerabilities. In addition, [40] provided a software tool, ESVD [30], to analyze code for vulnerabilities.

We focus on a vulnerability evident in the unsafe http request in the code in Fig. 14 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.

As described in Sect. 2, we have a methodology to allow us to select a subset of all sequence diagrams $\mathcal{D}$ and to rank those diagrams by the number of methods present in the diagram such that $|\mathcal{D}| = k$ where k is defined by user query. Our methodology of selecting and filtering proceeds as follows. From all diagrams generated, we select $D \in \mathcal{D}$ (using $k_I = 1$) with the most messages adding D to our interesting set $\mathcal{D}_I$. For all remaining diagrams $\mathcal{D} \backslash \{D\}$, we `filter` each diagram in $\mathcal{D}$ for code objects and methods found in D. We repeat this process until k diagrams are acquired. This procedure thus selects diagrams by how much new information they add.

Our experiment with blueblog defined $k = 45$. Without any filtering, *StaticGen* generated 2800 diagrams from blueblog. 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. 15. This example shows that prioritizing novel information allows us to reduce a set of diagrams while retaining information about code paths that can possibly lead to a vulnerability. This reduced set of diagrams 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 queries focusing on interestingness and filtration ensures each diagram provides

```
public class Main {
   public static void main(String[] args) {
      BaseClass obj = new Derived();
      System.out.println(obj.overridden());
      System.out.println(obj.unique());
   }
}
public class BaseClass {
   public String overridden() { return "From Base"; }
   public String unique() { return "In Base Only"; }
}
public class Derived extends BaseClass {
   @Override public String overridden() { return
        super.func() + " Derived"; }
}
```

Fig. 16 Polymorphic code example

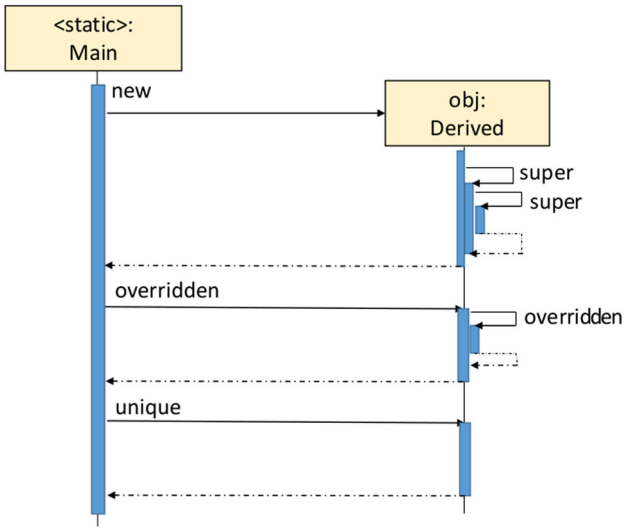

Fig. 17 *StaticGen* sequence diagram fragment of `SuperTestMain` from Fig. 16

novel information. The discarded diagrams, because they have fewer messages than the selected ones, and because they provide less novel information, would therefore less likely be the ones that solely contain/reveal the vulnerability. Hence, it is with low probability that a vulnerability is relegated to the set of discarded diagrams.

7.4 Analyzing polymorphic code paths

An important characteristic to object-oriented languages is dynamic dispatch through polymorphically defined methods. A sequence diagram, because it views the lifetimes and messages of different object instances, may not clearly explain at what level inheritance a particular method is actually handled. The UML language provides the ability to represent these situations using a "scenario box" that lists all possible alternatives. While listing all alternatives may properly convey operational semantics, it becomes cumbersome to users and obfuscates code in large class hierarchies. Soot, as an underlying component to *StaticGen* provides complete datatype information of each object: Datatype (`obj`) = `Derived` in the code from Fig. 16. We recognize that static analysis of code does have its limitations specifically with polymorphic code, but we wish to present how *StaticGen* handles basic polymorphism relying on Soot.

Sequence diagrams present object instances as having a single name and a single type, while an object instance passed to different methods may possess several names, and its interactions may be guided by any of the types in its inheritance ancestry. *StaticGen* maintains a list of the names and types used for a single object instance throughout a hypergraph traversal, and, when presenting that information as a sequence diagram, presents what it sees as the most expressive name and the most specific type. Generalizing the type of an object has a corresponding loss in precision. Our hypergraph traversal represents method calls with their complete names, including the package name, class name, method name, and argument type signature. While an argument can be made that sequence diagrams are not ideal for expressing more complex polymorphic interactions, hypergraph traversal can precisely handle all these interactions.

As a brief example of how *StaticGen* handles polymorphism, we consider the code in Fig. 16 and the corresponding sequence diagram in Fig. 17. In Fig. 16, a simple inheritance is established with `Derived` inheriting from `BaseClass`. In `main`, object `obj` is created with Datatype (*obj*) = `Derived` and two methods are subsequently called. Together these three method calls (`new`, `overridden`, and `unique`) are reflected clearly in the sequence diagram.

We consider these three calls in turn which correspond directly to the sequence diagram Fig. 17. Since constructors are not defined, yet exist in the bytecode, creation of object `obj` with `new` results in two subsequent self-calls to super-constructors in `BaseClass` and class `Object`. Next, `overridden` is called and handled in class `Derived`. We observe a self-call to `overridden` since `Derived.overridden` invokes a `super` call to `BaseClass.overridden`. Last, `unique` is handled entirely in `BaseClass`.

While *StaticGen* is effective for basic polymorphism among classes and hierarchies, *StaticGen* relies on Soot for pointer analysis. Consider a list of objects of type `BaseClass` (`List<BaseClass>`) which may contain both `BaseClass` and `Derived` objects. In the list structure, the contents become amorphous and the corresponding sequence diagrams present more generic information.

7.5 Diver case study

Sequence diagram generation is not a new idea. In this subsection, we compare *StaticGen* to an existing tool. Specif-

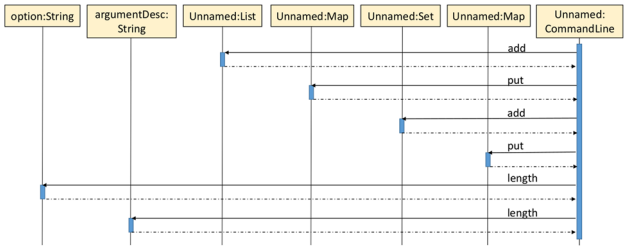

Fig. 18 *StaticGen* sequence diagram fragment of `addOption` from Fig. 19

```
public void addOption(String option, String
     argumentDesc, String description) {
   optionList.add(option);
   optionDescriptionMap.put(option, description);
   requiresArgumentSet.add(option);
   argumentDescriptionMap.put(option, argumentDesc);
   int width = option.length() + 3 +
        argumentDesc.length();
   if(width > maxWidth) {
      maxWidth = width;
   }
}
```

Fig. 19 `addOption` method from FindBugs [35]

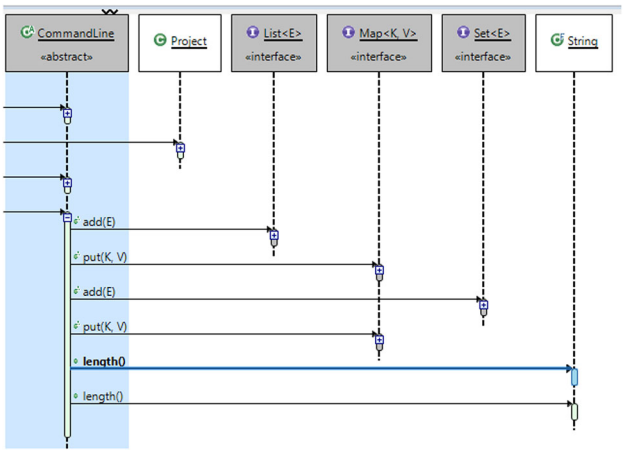

Fig. 20 FindBugs addOption diver output

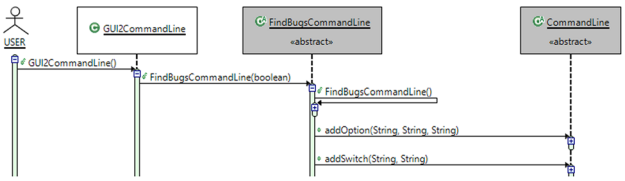

Fig. 21 FindBugs GUID2CommandLine diver output

ically, we compare a single run of our *StaticGen* to Diver [32], an eclipse plug-in which generates sequence diagrams from a trace of program execution. As described in related work (Sect. 8), other sequence diagram generation tools are described in the literature, but only a small subset of these tools are actually accessible. These UML Case tools [5,31,34,41,43] do not provide a query language that allows refinement in the way *StaticGen* provides. Dynamic tools [32] for generating sequence diagrams uncover sequence diagrams that actually occur during execution. It is therefore interesting to compare the effectiveness of a static tool like *StaticGen* with a dynamic tool like Diver. Our goal is to show that our static analysis tool compares well to a dynamic tool: static diagrams map to traces occur during dynamic execution. Our Diver tests were executed on version 0.5.0.201209240108, downloaded from the github repository [10] running Eclipse (Luna Service Release 1a 4.4.1) [16]. The target program was FindBugs [35]: a mature java source code base easily accessible for analysis with both tools. For full disclosure, it is important to state that the goal of our case study comparison should not be taken as a full feature comparison or a recommendation to use *StaticGen* over Diver. Our platform is an academic prototype while Diver is mature (undergone significant revision and bug fixing over several years). Additionally, we have extensive experience with our own tool and only basic experience with Diver. The working Diver example was found first, and then we attempted to replicate the same results using our tool (Fig. 18).

We focus on the `addOption` method in package `edu.umd.cs.findbugs.config.CommandLine` shown in Fig. 19. The sequence diagram in Fig. 18 depicts the output from Diver and its trace. With Diver, there does not appear to be an obvious way to save or export a specific diagram from a trace without providing the entire diagram. The Diver trace allows expansion of called methods to the desired level in the resulting sequence diagram. There are several similarities between the Diver diagram in Fig. 20 and *StaticGen* in Fig. 18; clearly the order of the methods are equivalent. *StaticGen* preserves more, but not all, variable names (`option` and `argumentDesc`). *StaticGen* also maintains individual objects thus differentiating method calls to `length` on distinct objects; similarly for the two calls to `put` on `Map`. It can be argued that the default Diver display is more compact; however, it is unclear in the expansion in Fig. 20 whether Diver differentiates method calls from distinct objects thus providing less accurate information.

We detail some differences between our approach and Diver. In package `edu.umd.findbugs.gui2`, we consider the constructor of class `GUI2CommandLine`. The diagram in Fig. 21 is the complete, allowable expansion of this flow through Diver starting at the class constructor. For the code in Fig. 22, *StaticGen* was able to visualize greater depth into the static class constructor for the class `Driver` (Fig. 23) in `edu.umd.cs.findbugs.gui2`. Generally, in comparison, *StaticGen* can offer users a view of object creation from any level; *StaticGen* produced 55 total potential execution path diagrams which we could query and sort.

```
public class Driver{
  private static long START_TIME =
       StartTime.START_TIME;
  private static final String USAGE =
       Driver.class.getName() ...
  private static GUID2CommandLine commandLine = new
       GUI2CommandLine();
  ...
  }
```

Fig. 22 Driver class static code

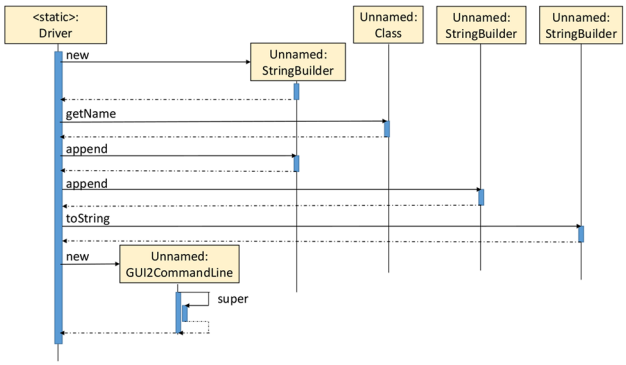

Fig. 23 *StaticGen* output of `GUID2CommandLine` from Fig. 22

8 Related work

In [22,23,29], 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 [34] generate sequence diagrams directly from application runtime while [34] uses the Eclipse IDE [15] to reverse engineer all or part of a stack trace. Similarly, [17] divides a long dynamic trace of a Java program into a series of smaller diagrams culminating in a sequence diagram. Finally, [47] describes an approach for generating sequence diagrams dynamically using a k-tail merging algorithm that merges the collected traces. The goal of merging by [47] 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 [43] 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 [41] and Visual Studio [31] 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) [18] employs a combination of static and trace-based information and communicates the overall importance of visualizing source code. Tonella and Potrich [42] described static extraction of UML sequence diagrams from C++ code using partial analysis and focusing, but do not perform analysis of intra-procedural flow of control. The CPP2XMI tool [19] processes XMI into sequence diagrams with no means of user-based refinement as with *StaticGen*. I2SD [36] is a static generation tool that leverages metadata through interceptors, whereas our technique does not rely on such information. The RED tool [37,38] 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 [37]. In total, our approach seeks to empower the user by supporting query-based refinement over the set of all sequence diagrams. Every sequence diagram that corresponds to a trace of a program will be generated by *StaticGen*. However, a purely static tool like *StaticGen* may generate interactions that may never get executed. Any critical interaction (after refinement) uncovered by *StaticGen* can be dynamically validated by a test case extracted from the resulting sequence diagram.

In [27,28] 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.

Graph/hypergraph exploration techniques have been used previously for synthesis in other domains. Hypergraph pebbling techniques have been used for problem and solution generation for High School Geometry [2,3]. Graph exploration techniques have been used for generating target-focused libraries for drug discovery in [33].

9 Conclusions and future work

This paper describes a framework for static generation of sequence diagrams[5] 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. Future work

[5] Scenario diagrams.

will address how limitations of our tool manifest themselves. This includes identification and elimination of impossible paths, formally handling immediate and non-immediate recursion, loops, as well as performing further case study analyses on non-Android codebases. In addition, we will explore the use of *StaticGen* in understanding the impact of service compositions [45,46] in Service-Oriented Computing.

References

1. Alur, R., Etessami, K., Yannakakis, M.: Inference of message sequence charts. In: Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4–11, 2000, pp. 304–313 (2000)
2. Alvin, C., Gulwani, S., Majumdar, R., Mukhopadhyay, S.: Synthesis of geometry proof problems. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27–31, 2014, Québec City, Québec, Canada, pp. 245–252 (2014)
3. Alvin, C., Gulwani, S., Majumdar, R., Mukhopadhyay, S.: Synthesis of solutions for shaded area geometry problems. In: Proceedings of the Thirtieth International Florida Artificial Intelligence Research Society Conference, FLAIRS 2017, Marco Island, Florida, USA, May 22–24, 2017, pp. 14–19 (2017)
4. Architexa.com: Introduction to architexa | sequence diagram generation (2015). http://www.architexa.com/support/videos/sequence-diagrams. Accessed 2 Feb 2017
5. Bedrin, D.: jtracert (2015). https://code.google.com/p/jtracert/. Accessed 2 Feb 2017
6. Beizer, B.: Software Testing Techniques, 2nd edn. Van Nostrand Reinhold, New York (1990)
7. Berge, C.: Graphs and Hypergraphs. North-Holland Mathematical Library, vol. 45. Elsevier, Amsterdam (1989)
8. Buren, R.: BlueBlog. https://sourceforge.net/projects/blueblog/. Accessed 2 Feb 2017
9. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
10. Chisel Group: Diver github (2016). https://github.com/thechiselgroup/Diver. Accessed 2 Feb 2017
11. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
12. XDA Developers: List of all open-source Android apps (2013). http://forum.xda-developers.com/showthread.php?t=2124002. Accessed 2 Feb 2017
13. 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)
14. Harel, D., Thiagarajan, P.: UML for Real: Design of Embedded Real-Time Systems: Message Sequence Charts, 1st edn. Kluwer Academic Publishers, Dordrecht (2003)
15. E.F. Inc. Eclipse (2015)
16. E.F. Inc. Eclipse luna (2016)
17. Ishio, T., Watanabe, Y., Inoue, K.: AMIDA: a sequence diagram extraction toolkit supporting automatic phase detection. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10–18, 2008, Companion Volume, pp. 969–970 (2008)
18. Jerding, D.F., Stasko, J.T., Ball, T.: Visualizing interactions in program executions. In: Pulling Together, Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA, May 17–23, 1997, pp. 360–370 (1997)
19. 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: 13th Working Conference on Reverse Engineering (WCRE 2006), 23–27 October 2006, Benevento, Italy, pp. 297–298 (2006)
20. Koskimies, K., Mössenböck, H.: Scene: using scenario diagrams and active text for illustrating object-oriented programs. In: Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, March 25–29, 1996, pp. 366–375 (1996)
21. Kroening, D.: goto-cc—a c/c++ front-end for verification (2015). http://www.cprover.org/goto-cc/. Accessed 2 Feb 2017
22. Kumar, S., Khoo, S., Roychoudhury, A., Lo, D.: Mining message sequence graphs. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21–28, 2011, pp. 91–100 (2011)
23. Kumar, S., Khoo, S., Roychoudhury, A., Lo, D.: Inferring class level specifications for distributed systems. In: 34th International Conference on Software Engineering, ICSE 2012, June 2–9, 2012, Zurich, Switzerland, pp. 914–924 (2012)
24. Labiche, Y., Kolbah, B., Mehrfard, H.: Combining static and dynamic analyses to reverse-engineer scenario diagrams. In: 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22–28, 2013, pp. 130–139 (2013)
25. Leucker, M., Madhusudan, P., Mukhopadhyay, S.: Dynamic message sequence charts. In: Proceedings of the FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science, 22nd Conference Kanpur, India, December 12–14, 2002, pp. 253–264 (2002)
26. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium—Volume 14, SSYM'05, p. 18. USENIX Association, Berkeley (2005)
27. Lo, D., Maoz, S.: Mining scenario-based triggers and effects. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15–19 September 2008, L'Aquila, Italy, pp. 109–118 (2008)
28. Lo, D., Maoz, S.: Mining hierarchical scenario-based specifications. In: ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16–20, 2009, pp. 359–370 (2009)
29. Lo, D., Maoz, S., Khoo, S.: Mining modal scenario-based specifications from execution traces of reactive systems. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5–9, 2007, Atlanta, Georgia, USA, pp. 465–468 (2007)
30. Luciano Sampaio: Early Security Vulnerability Detector. https://marketplace.eclipse.org/content/early-security-vulnerability-detector-esvd. Accessed 16 Oct 2016
31. Msdn.microsoft.com: Visualize code on sequence diagrams (2015). https://msdn.microsoft.com/en-us/library/ee317485.aspx

32. Myers, D., Storey, M.-A.: Using dynamic analysis to create trace-focused user interfaces for ides. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pp. 367–368. ACM, New York (2010)
33. Naderi, M., Alvin, C., Ding, Y., Mukhopadhyay, S., Brylinski, M.: A graph-based approach to construct target-focused libraries for virtual screening. J. Cheminform. **8**(1), 14:1–14:16 (2016)
34. Objectaid.com: UML explorer (2015). http://www.objectaid.com/sequence-diagram. Accessed 2 Feb 2017
35. Pugh, B., Loskutov, A.: Findbugs, find bugs in java programs (2015). http://findbugs.sourceforge.net/index.html. Accessed 2 Feb 2017
36. Roubtsov, S.A., Serebrenik, A., Mazoyer, A., van den Brand, M.G.J., Roubtsova, E.E.: I2SD: reverse engineering sequence diagrams enterprise Java beans from with interceptors. IET Softw. **7**(3), 150–166 (2013)
37. Rountev, A., Connell, B.H.: Object naming analysis for reverse-engineered sequence diagrams. In: 27th International Conference on Software Engineering (ICSE 2005), 15–21 May 2005, St. Louis, Missouri, USA, pp. 254–263 (2005)
38. Rountev, A., Volgin, O., Reddoch, M.: Static control-flow analysis for reverse engineering of UML sequence diagrams. In: Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05, Lisbon, Portugal, September 5–6, 2005, pp. 96–102 (2005)
39. Sable Research Group: Soot: a framework for analyzing and transforming java and android applications (2015). http://sable.github.io/soot/. Accessed 2 Feb 2017
40. Sampaio, L., Garcia, A.: Exploring context-sensitive data flow analysis for early vulnerability detection. J. Syst. Softw. **113**, 337–361 (2016)
41. Soyatec.com: Soyatec–sequence diagram generation (2015). http://www.soyatec.com/euml2/features/eUML2%20Modeler/. Accessed 2 Feb 2017
42. Tonella, P., Potrich, A.: Reverse engineering of the interaction diagrams from C++ code. In: 19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22–26 September 2003, Amsterdam, The Netherlands, pp. 159–168 (2003)
43. Visual-paradigm.com: Reverse engineering sequence diagram from java source code (2015). https://www.visual-paradigm.com/tutorials/seqrev.jsp. Accessed 2 Feb 2017
44. Wikipedia: List of free and open-source android applications (2015). http://en.wikipedia.org/wiki/List_of_free_and_open-source_Android_applications. Accessed 2 Feb 2017
45. Yau, S.S., Davulcu, H., Mukhopadhyay, S., Huang, D., Gong, H., Singh, P., Gelgi, F.: Automated situation-aware service composition in service-oriented computing. Int. J. Web Serv. Res. IJWSR **4**(4), 59–82 (2007)
46. Yau, S.S., Davulcu, H., Mukhopadhyay, S., Huang, D., Yao, Y.: Adaptable situation-aware secure service-based (as/sup 3/) systems. In: Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), pp. 308–315. IEEE (2005)
47. Ziadi, T., da Silva, M.A.A., Hillah, L., Ziane, M.: A fully dynamic approach to the reverse engineering of UML sequence diagrams. In: 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27–29 April 2011, pp. 107–116 (2011)