International Journal of Information System Modeling and Design
Volume 14 • Issue 1

Automatic Test Sequence Generation and Functional Coverage Measurement From UML Sequence Diagrams

Nazım Umut Ekici, Izmir Institute of Technology, Turkey
Tugkan Tuglular, Izmir Institute of Technology, Turkey*
https://orcid.org/0000-0001-6797-3913

ABSTRACT

Sequence diagrams define functional requirements through use cases. However, their visual form limits their usability in the later stages of the development life cycle. This work proposes a method to transform sequence diagrams into graph-based event sequence graphs, allowing the application of graph analysis methods and defining graph-based coverage criteria. This work explores these newfound abilities in two directions. The first is to use coverage criteria along with existing tests to measure their coverage levels, providing a metric of how well they address the scenarios defined in sequence diagrams. The second is to use coverage criteria to automatically generate effective and efficient acceptance test cases based on the scenarios defined in sequence diagrams. The transformation method is validated with over eighty non-trivial projects. The complete method is validated through a non-trivial example. The results show that the test cases generated with the proposed method are more effective at exposing faults and more efficient in test input size than user-generated test cases.

KEYWORDS

Acceptance Testing, Event Sequence Diagram, Functional Coverage, Sequence Diagram, Test Case Generation

INTRODUCTION

Unified Modelling Language (UML) sequence diagrams graphically describe system component interactions, arranged in time sequence (OMG Unified Modelling Language, 2017). The temporal nature of sequence diagrams makes them suitable for capturing functional scenarios. System components in sequence diagrams can vary from higher-level entities, such as users and software systems, to lower-level entities, such as individual objects and methods, allowing the specification of functional scenarios at all levels of design. Although sequence diagrams are widely used to describe both higher and lower-level functional scenarios, their informal

DOI: 10.4018/IJISMD.332865

*Corresponding Author

and graphical format limits their usability in the software development life cycle. This work aims to increase their usability by proposing a method to utilize sequence diagrams in test sequence generation and functional coverage level measurement. A test sequence is an ordered list of elements, where elements are in the alphabet of the system's set of states and events. They are used to express the desired transitions for the system under test. Test sequences are materialized by test inputs, where a test input is an ordered list of elements from the alphabet of possible system inputs. For a deterministic system in a given state, application of a test input results in a certain test sequence.

The proposed method shows that sequence diagrams can be transformed into equivalent formal and graph-based event sequence graphs (ESGs) (Belli, 2006), and their equivalent ESGs can be utilized to define formal coverage criteria for testing of functional scenarios and allow measurement of coverage levels of such criteria. This is one of the novelties presented in this work; another is the process of test sequence generation to satisfy specific coverage criteria. The proposed method utilizes existing methods for test sequence generation from ESGs on a per-sequence diagram approach to achieve a set of efficient and effective test inputs. For evaluation, the proposed method is applied to the one and only found public project with UML sequence diagrams and executable acceptance test cases. The novelty of this work enabled the evaluation to compare the test sequences that are generated by users to the ones that are automatically generated via a transformation from sequence diagrams in terms of effectiveness and efficiency.

This paper makes the following contributions: (1) Method – The proposed method transforms sequence diagrams into equivalent ESGs and utilizes them to define test coverage criteria for functional scenarios and to measure coverage levels of test inputs for these criteria. Equivalent ESGs are also used to generate efficient and effective test sequences to satisfy defined coverage criteria. This work validates the proposed transformation and test sequence generation method on more than 80 public repositories for a wide range of different types of projects. Additionally, this work compares the test sequences generated by the proposed method with the ones generated by a user on an existing project, contemplating its applicability in practice. (2) Tool – Two tools were developed to implement the method explained in (1). The tools are shared in public repositories.

The first part of the proposed approach, namely the transformation of sequence diagrams and generation of the test sequences, imposes no requirement on the inputs, other than they be in the PlantUML syntax (as it is a widely used format). This is further demonstrated by the newly added validations on over 80 projects. The second part of the approach, namely the measurement of coverage levels, requires function and class names used in the sequence diagrams and the code itself to match, to be able to map them accurately. Ideally that should be the case (as the opposite would imply a mismatch between the design and implementation, which would result in lower coverage levels); however how well this is followed in industrial projects is a valid point. The scope of the second part of the approach focuses on function-call-based coverage criteria (e.g., call coverage or call-pair coverage), as this pairs well with sequence diagrams using functions as their smallest and fundamental unit (within message bodies), and with parameter types and values often omitted. In the future, research will extend the approach with coverage types involving parameter values and perhaps utilizing frame guard expressions.

The manuscript is organized as follows: Section 2 presents the fundamentals of the concepts used throughout this work. Section 3 explores the related works in the literature. Section 4 and 5 explain the proposed method. Section 6 describes the developed software tools implementing the proposed method. Section 7 tests and evaluates the proposed method. Threats to validity are discussed in Section 8, and Section 9 concludes the paper.

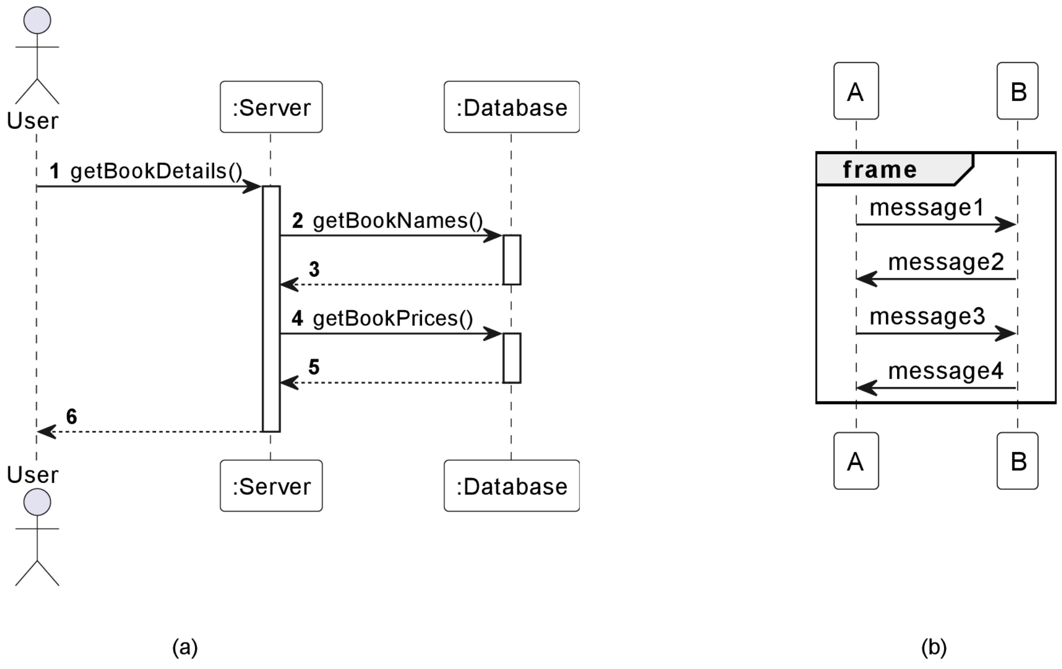
OVERVIEW

Sequence Diagram

UML sequence diagrams present the dynamic behavior of a system for a single use-case (OMG Unified Modelling Language, 2017). They provide a visual and easy-to-understand representation of the described use case. They show the interactions between objects in the order they occur. Figure 1a depicts a sequence diagram. Every object is represented as a vertical lane, called a *lifeline*. Each lifeline's beginning is labeled with its corresponding object. Interactions between the objects are shown in the form of solid, horizontal, and directional arrows between objects and are called *messages*. Message names are written above the corresponding message arrows. Message arrows begin at the source object and terminate at the target object. Optionally, responses to messages are shown as arrows with dashed lines.

The Unified Modelling Language Specification (2023) introduced a new notation element named *frame*. Messages in a sequence diagram can be grouped together in frames, as shown in Figure 1b. Frames are shown as boxes with related messages enclosed in them. The frame's name is written on the top left corner of the box. Multiple frames can be nested to represent more complex interactions increasingly. Sequence diagram specification reserves some frame names to mark frames with special semantics, such as *opt*, *loop*, and *alt*. These frame types have a *guard expression*, denoted at the top right corner of the frame, expressing the condition for which the statements in the frame will occur. Opt frames have a guard expression, denoted next to the frame's name. The interactions within the opt frame will occur only if the guard expression evaluates to true at this point in the interaction. Like the opt frame, the loop frame also has a guard expression. Interactions within the loop frame will occur and keep occurring as long as the guard expression evaluates to true. The alt frame has multiple sections, each with its guard expressions. Only one of the guard expressions within the alt

Figure 1. Sequence diagram and frame



frame can evaluate to true at a time, and the interactions within the section with its guard expression evaluating to true will occur. The introduction of frame notation has increased the expressive power of the sequence diagrams, enabling them to model more complex behaviors.

Event Sequence Graph

Event sequence graphs (ESG) (Belli, 2001) are used to define systems in a simple graph format. ESGs are directed graphs where each vertex represents an event within the system. An edge from vertex *v* to vertex *v'* exists if and only if the event *v'* can occur following the event *v*. To mark the entry and exit vertices of the ESG, two special pseudo-vertex "*[*" and "*]*" are used, where the "*[*" vertex denotes the initial state of the system and the "*]*" vertex denotes the final state. ESG is a directed graph where any walk of the graph describes a valid event sequence for the system it models (Belli, 2012). Any walk starting at the "*[*" vertex and ending at the "*]*" vertex is a *complete event sequence* (Belli, 2006).

To model sub-flows of the system, ESGs can be nested. Instead of representing an event, a vertex can represent another ESG, dubbed a *sub-ESG* (Belli, 2005). Once the modeled system reaches the sub-ESG vertex, sub-ESG assumes the control of the flow, starting from its "*[*" vertex. Once the "*]*" vertex of the sub-ESG is reached, the control of the flow is transferred back to the parent ESG.

RELATED WORK

Lübke (2006) first introduces a meta-model for UML use cases written in the Cockburn template (Cockburn, 1998). Then the work proposes a method to transform such UML use cases into event-driven process chains (EPC) (Aalst, 1999). The proposed method takes multiple UML use cases, transforms them into multiple EPC fragments, and then merges these fragments into a single EPC diagram, allowing the representation of a global control flow. The proposed method is implemented and evaluated on an existing project.

Similarly, Lübke et al. (2008) propose a method to transform sets of UML use cases written in the Cockburn template into Business Process Modelling Notation (Dijkman et al., 2008).

Gherkin (Gherkin Reference, 2023) is a specification language that is widely used for behavior driven development. Alferez et al. (2019) propose a method to transform specifications represented as UML activity diagrams and use case diagrams into Gherkin scenarios. The transformed Gherkin scenarios are then used as acceptance criteria, bridging the gap between requirements and verification.

Gherkin can also be used to automate the generation of tests. Guiterrez et al. (2017) propose a method to transform UML use case diagrams into Gherkin scenarios. The proposed method utilizes the test generation ability of the Gherkin language to generate tests from UML use case diagrams automatically.

Sarma et al. (2007) propose a method to transform sequence diagrams adhering to the former UML specification (Roques, 2003), collectively known as the UML 1.x, into a graph-based representation named Sequence Diagram Graph (SDG). They propose using the transformed SDGs to generate test sequences satisfying the path-coverage criterion. Similarly, Bernardi et al. (2002) propose a method to transform UML 1.x sequence diagrams into Petri Nets.

Swain et al. (2010) propose a method to transform sequence diagrams adhering to the UML specification of 2.0 into concurrent control flow graphs (CCFG). The transformed CCFG is then used to generate full predicate coverage test sequences.

Jena et al. (2015), Sumalatha and Raju (2013), and Hoseini and Jalili (2014) propose similar methods to transform sequence diagrams into graph-based intermediate forms. Their methods generate test sequences for path-coverage and vertex-coverage. Each use genetic algorithms to improve test sequence efficiency and do not guarantee optimal results. Similarly, Chandnani et al. (2017) transform sequence diagrams into a graph-based method to generate test sequences for vertex-coverage. Then they propose to use particle swarm optimization algorithm to improve efficiency, with no guarantee on optimal results. And finally, Li et al. (2007), Cartaxo et al. (2007), Priya and Malarchelvi (2013), Lei and Lin (2008), and Zhang et al. (2016) all similarly propose a method to transform sequence diagrams into graph-based forms to generate test sequences with no consideration for test sequence efficiency. For generated test sequences vertex-coverage criterion is satisfied by the method proposed by Li et al. (2007). The methods proposed by Cartaxo et al. (2007) and Lei and Lin (2008) satisfy edge-coverage. The methods proposed by Zhang et al. (2016) and Priya and Malarchelvi (2013) satisfy basic-path coverage.

Mallick et al. (2014) and Khandai et al. (2011) utilize resource dependency graphs and UML activity diagrams along with sequence diagrams, in order to create test sequences for concurrent systems. Generated test sequences are able to cover possible deadlock scenarios and satisfy activity-path coverage criterion.

The novel method presented in this work transforms UML sequence diagrams into an existing and well-established graph-based representation, namely ESG. Additionally, the proposed method is able to generate minimal test sequences to satisfy arbitrary-length path coverage criteria. The state-of-art methods in the literature are able to generate test sequences satisfying a more limited range of criteria (e.g. vertex or edge coverage), with no guarantees on generated test sequences being minimal. The proposed method also utilizes the graph representation that is driven from sequence diagrams to measure coverage levels for acceptance tests. This enables the specifications in the form of sequence diagrams to be utilized in defining an exit criterion for testing. The proposed transformation method is also applied to and validated with over 80 real-life and public projects, rather than an example sequence diagram devised to demonstrate the method. Furthermore, the complete proposed method is applied to an existing and complete real-life project, proving a unique opportunity to compare test sequences written by a user and test sequences generated by transforming and analyzing sequence diagrams in terms of effectiveness and efficiency. Lastly, the presentation of two publicly available tools, working with widely used input formats, enable industry professionals to apply the proposed method to their work.

PROPOSED METHOD

The proposed method transforms a UML Sequence Diagram (SD) into an ESG by a series of transformation rules, enabling coverage-based test sequence generation by ESGs.

For the transformation of a sequence diagram to an equivalent ESG, all sequence diagram elements other than messages and frames can be ignored. Sequence diagrams have many other elements, such as activation and lifelines. However, for the purpose of generating test sequences, these elements are irrelevant, as the only concern is which events may occur and in which order. This simplification leads to the simplified sequence diagram model shown in Figure 2. Frames consist of other frames and messages. Three particular types of frames require special treatment, namely *opt*, *alt*, and *loop* frames. The sequence diagram itself is a frame of the type *sd*. Messages have two participants: a *source* and a *target*.

Each element of the simplified sequence diagram can be converted into an equivalent ESG vertex using a set of transformation rules. There are five different transformation rules for successive messages in a frame, opt frames, alt frames, loop frames and nested frames. The remainder of the section describes these transformations.

Transformation of Messages in a Frame

Messages in a frame denote an ordered set of events that are permitted to occur one after the other. Therefore, they can be chained in the same order they appear in the frame to create an event chain. Each event in the ESG is named with the target of its corresponding message, followed by the message body, formatted in the object invocation syntax. The first message's event is connected to the entry vertex of the ESG, while the last message's event is connected to the exit vertex. A sample conversation is presented in Figure 3.

Figure 2. Simplified sequence diagram meta-model as a class diagram

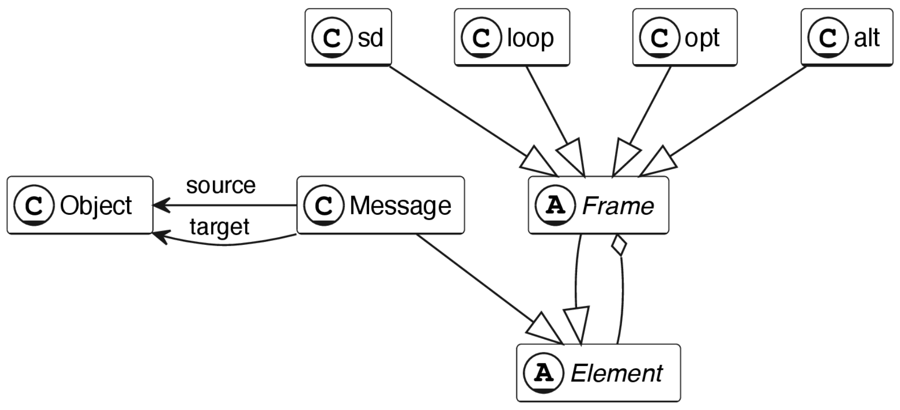
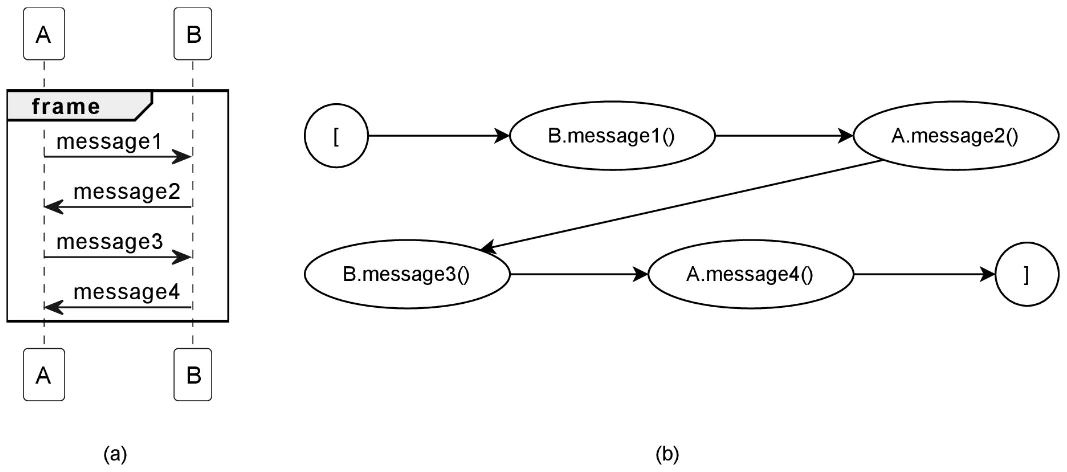


Figure 3. (a) Frame with four messages and (b) its ESG equivalent



Transformation of Opt Frame

Opt is a special frame preceded by a guard clause. Its messages are executed only if its guard clause is satisfied. Messages in the body of an opt frame can be transformed into an ESG segment as described in Section 4.1, as similarly, body messages denote an ordered set of events that can follow each other. Opt frame's ESG equivalent has two paths from start to end vertex. One path for the case where the guard clause is satisfied and the body of the opt frame is executed. A second path is for the case where the guard clause is not satisfied, and none of the events in the frame occurs. Figure 4 presents such a transformation.

Transformation of Alt Frame

An alt frame has multiple fragments, each with its own guard clause. Only the fragment whose guard is true will execute. Its ESG equivalent has as many paths from start to end vertex as there are fragments. Each path starts with a guard clause, followed by an event chain corresponding to the messages in the related fragment. Figure 5 shows a sample transformation.

Figure 4. (a) Opt frame with two messages and (b) its ESG equivalent

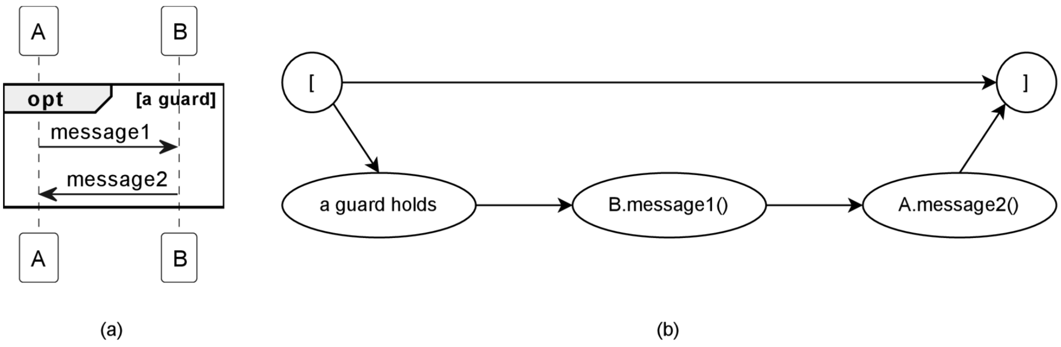
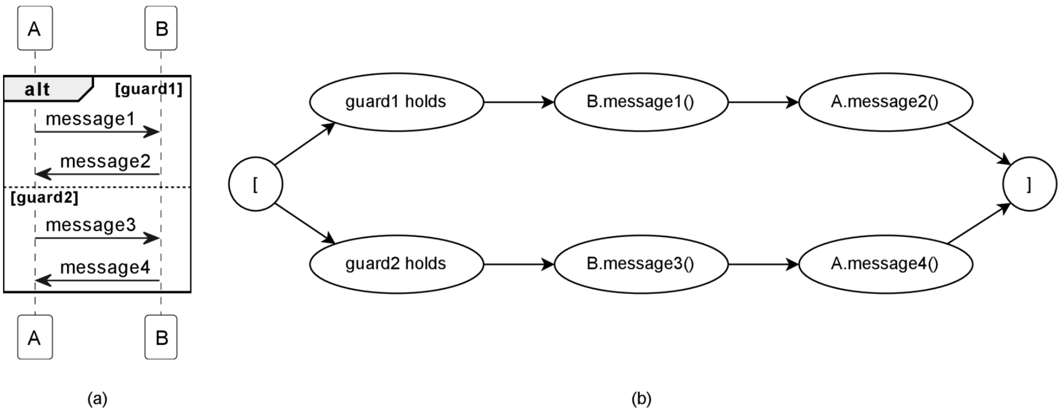


Figure 5. (a) Alt frame with two fragments with two messages each and its ESG



Transformation of Loop Frame

A loop frame's messages are repeated as long as its guard clause is true. Its ESG equivalent has two simple paths from start to end vertex: one for the case where the loop guard holds and another for the case it does not hold. The guard holds event is followed by the messages in the frame's body. After the messages, either the messages can be repeated or not, depending on whether or not the guard holds. Figure 6 shows a sample transformation.

Transformation of Nested Frames

Along with messages, sequence diagram frames can contain other frames. For frames containing other frames, the inner frames are converted to a single message on the sequence diagram, and to an associated ESG, following the previously given set of rules. For the transformation of the frame, all other messages are treated the same for the transformation of the frame. They are represented by a simple ESG vertex named after the message. The messages representing inner frames are also represented by a single ESG vertex. Instead of a simple ESG vertex, they are represented by a parent vertex containing the message's associated ESG. For multiple nested frames, this rule must be applied starting from the innermost frame.

Figure 7 shows the transformation for a nested frame. A sequence diagram with an opt frame nested in a loop frame is shown in (a). The opt frame's equivalent ESG according to the transformation

Figure 6. (a) Loop Frame with Two Messages and (b) its ESG Equivalent

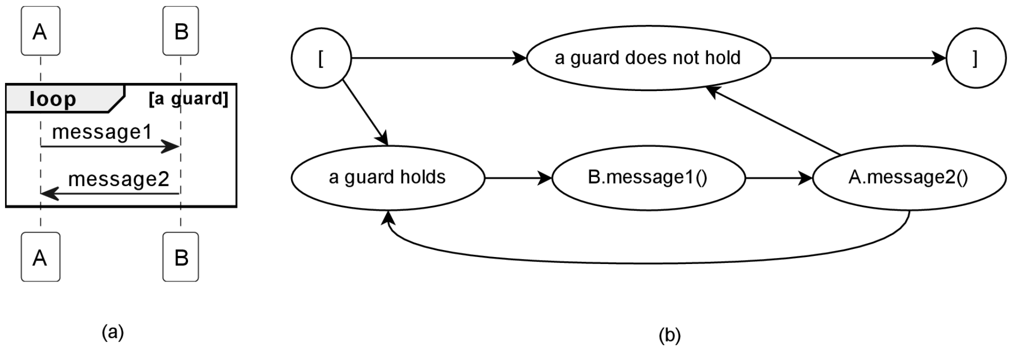
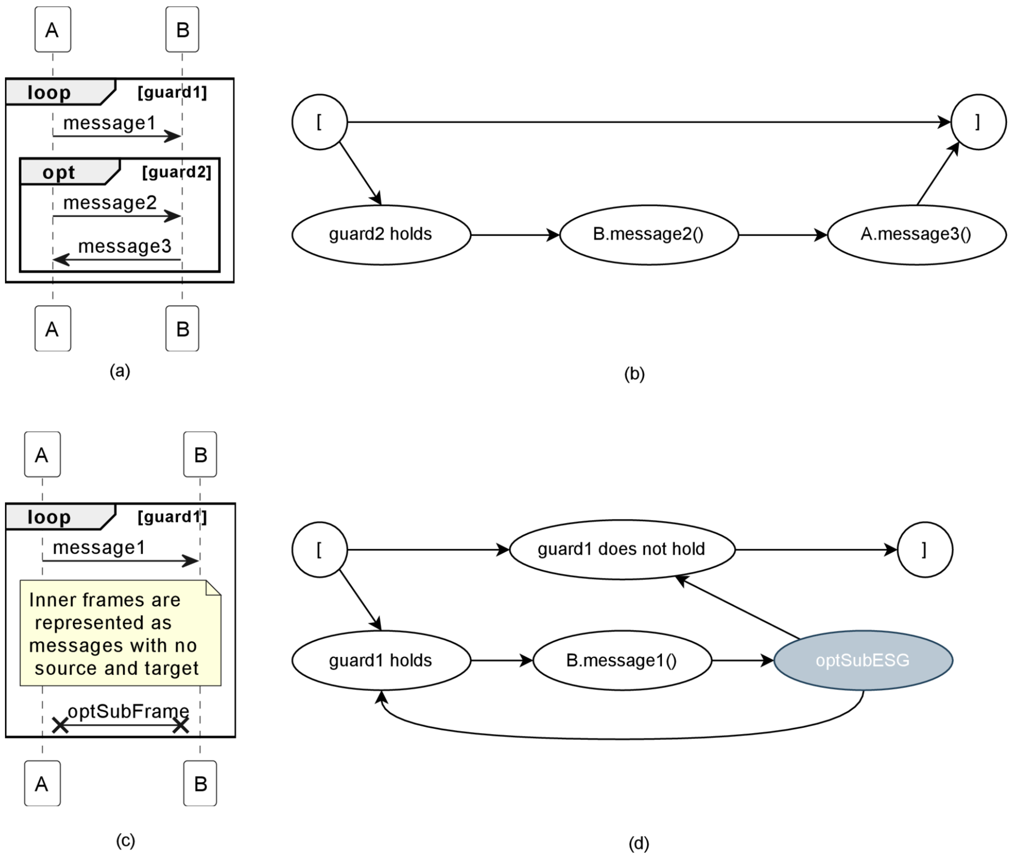


Figure 7. Transformation process for a sequence diagram with a nested frame



in Section 4.2 is given in (b). After all inner frames are treated, the intermediary sequence diagram contains only regular messages and messages representing inner frames, as shown in (c). Finally, the intermediary sequence diagram is transformed into an ESG. The resulting ESG (d) has parent ESG vertices corresponding to each inner frame.

A loop frame with a nested opt frame (a), ESG equivalent of the inner opt frame (b), intermediary sequence diagram with inner frames replaced by messages (c), ESG equivalent of the outer loop frame. The parent ESG vertex is shown with a darker shade (d).

Repeated application of the above rules, starting from the innermost frames, results in a hierarchy of ESGs, closely reflecting the hierarchy of its source sequence diagram. The highest level ESG corresponds to the top-level frame in the sequence diagram. For each frame nested in the outer frame, its corresponding ESG has a sub-ESG.

The algorithm describing all of the transformations defined in this section is given in Algorithm 1. The time complexity of this recursive operation can be analyzed as follows. The *initial_esg* step takes a constant amount of time for each frame, appending guard statements in a frame-specific manner. The for loop over the elements of the frame is iterated *N* times across all recursions, where *N* is the number of elements (i.e., messages and other frames) in the frame. The *esg.append* step within the for loop also takes a constant time for each element. Combining the expressions for the number of iterations with the complexity of each iteration yields *$\theta(N)\theta(1)$*, simplifying to *$\theta(N)$*. The method described in this section is implemented with Java and provided as a publicly hosted repository (*SequenceDiagram2ESG, 2023)*.

Algorithm 1 shows the process to transform sequence diagram frames with nested frames into their equivalent ESGs.

```
Input frame: A sequence diagram frame
Output esg: equivalent of event sequence diagram
procedure TRANSFORM_FRAME_TO_ESG
        esg ← ∅
        if frame.type ∈ {alt, opt, loop} then       // initialize the esg with
                esg ← initial_esg(frame.type)  // rules defined in
        endif                                       // section 4.2 - 4.4
        for element ∈ frame.elements do
                if element.is_frame() then
                        sub_esg ← TRANSFORM_FRAME_TO_ESG(element)
                        esg.append(sub_esg)
                else
                        esg.append(element)
                endif
        endfor
        return esg
endprocedure
```

FUNCTIONAL COVERAGE LEVEL MEASUREMENT WITH SEQUENCE DIAGRAMS

Sequence diagrams offer a simple and visual representation to describe system functionality via certain use cases. Testing activities for verification of such functionalities can be conducted using the sequence diagram as a reference. However, the informal structure of the sequence diagrams does not allow defining quantitative coverage criteria, which limits their usability in testing. A sequence diagram offers numerous (and possibly infinite) paths from which a test execution may choose. In the absence of a coverage criteria, choosing a set of test paths among the set of all possible paths becomes a tedious and arbitrary process with no clearly defined goals. The absence of a coverage criteria and its associated measurable metrics makes it difficult to design tests and measure their effectiveness and efficiency.

ESGs, on the other hand, with their graph-based structure and formal nature, can be used with any coverage criteria that apply to graphs (Ammann & Offutt, 2016). Therefore, transforming sequence diagrams to a set of ESGs enables the definition of coverage criteria on a graph representation of functional scenarios.

As the messages in the sequence diagrams are method calls, the derived ESG vertices are also labeled by method calls. Applying a coverage criterion on the derived ESGs results in a set of test requirements in which each element consists of names of method calls. This relation to the code under test enables automated measurement of coverage levels of a test set by comparing the test requirements against the actual method invocations during the execution of the test set. For example, vertex-pair coverage criteria require each reachable path of length 1 to be in test requirements. More formally $TR = \{[v, v'] \mid (v, v') \in E \wedge (v is reachable)\}$. Applying vertex-pair coverage to the ESG in Figure 6 results in the test requirements set of:

$$TR = \left\{ \begin{bmatrix} [, guard1\ holds, & \begin{bmatrix} guard1\ does\ not\ hold, guard1\ holds, B.message1(\ ), \\ B.message1(\ ), A.message2(\ ), A.message2(\ ), guard1\ does\ not\ hold, \\ A.message2(\ ), guard1\ hold, guard1\ does\ not\ hold \end{bmatrix} \end{bmatrix} \right\}$$

where each member of the set is an ordered pair of method calls.

A single test run with the trace "[, guard1 holds, B.message1(), A.message2, guard1 does not hold,]" satisfies 5 of the 6 test requirements, resulting in the coverage level of 83%.

For a set of sequence diagrams describing the system's functionality, the set of test requirements is the union of the individual test requirement sets belonging to each sequence diagram. In the case of multiple test runs with different sets of test inputs, if a test requirement is satisfied in any of the test runs, it is considered to be satisfied. This approach also helps highlight the relevance of each test to each sequence diagram since it is possible to measure their coverage levels of functional scenarios individually. The algorithm describing the approach for calculating arbitrary length path coverage level is given in Algorithm 2.

Algorithm 2 shows the process to measure coverage level for n-length path criterion coverage level.

```
Input esgs: Set of ESGs describing the system
Input traces: execution traces generated by test runs
Input path_length: path length for the coverage criterion
Output covered_paths: set of paths from esgs with length = path
         length visited in traces
Output not_covered_paths: set of paths from esgs with length = path
         length not visited in traces
procedure CALCULATE_N_LENGTH_PATH_COVERAGE
         covered_paths ← ∅
         not_covered_paths ← ∅
         for esg ∈ esgs do
                  paths ← esg.paths_with_length(path_length)
                  for path ∈ paths do
                           if traces.contain(path) then
                                    covered_paths ← covered_paths ∪ {path}
                           else
                                    not_covered_paths ← not_covered_
                                    paths ∪ {path}
                           endif
                  endfor
         endfor
         return covered_paths, not_covered_paths
endprocedure
```

The time complexity of this algorithm can be calculated as the following. Calculating the *paths* with a given length L for a graph with N vertices takes $O(N^L)$ time with a depth-first search. The inner for loop will be iterated as many times as there are paths with length L. A graph with N vertices has k^L paths with length L, where k is the average number of edges per vertex. With the proposed transformation method, all messages in the sequence diagram are connected with a single edge in the ESG, since they represent a non-conditional and linear flow. Only ESG vertices with more than one edge are introduced by the frame, where a decision is made due to a guard statement. Therefore, for an ESG generated from a sequence diagram with M guard statements, there are $2M + N$ edges, resulting in an average number of edges per vertex of $1 + 2M/N$. As a result, the inner loop is iterated $(1 + 2M/N)^L$ times. The search for a path with length L in a trace with length T can be done with $\theta(L) + \theta(T)$ time complexity with Knuth-Morris-Pratt algorithm (Knuth et. al., 1997). This complexity can be simplified to $\theta(T)$, since the $T \gg L$. These terms yield the total time complexity of the iterations as $O(T(1 + 2M/N)^L)$. Combining this with the cost of calculating paths gives the final time complexity of $O(N^L) + O(T(1 + 2M/N)^L)$. While the term N in the denominator may look counter-intuitive, M, the number of guard statements scales with N. Introducing a new term f as the average number of messages per frame gives the relation $f=N/(2M)$. This term can be substituted in the previous expression to give $O(N^L) + O(T((1 + 1/f)^L)$. The relation shows that as the number of messages in the frame decreases, complexity increases due to a higher number of frames and, consequently, a higher number of conditional paths.

The method described in this section is implemented with Java and provided as a publicly hosted repository (*ESGCoverageMeasurer, 2023).*

As the sequence diagrams can be used to measure functional coverage levels of existing test inputs, they can also be utilized to create test inputs to satisfy a chosen functional coverage criterion. Belli and Budnik (2007) proposed a method to generate test sequences to satisfy a coverage criterion for a system defined in the form of an ESG with the smallest possible number of test statements and test runs. Even though this approach would still require manual derivation of test inputs that would trigger the generated test sequences, having a desired test sequence to guide the user would reduce the complexity of the test input generation. Moreover, as the method in Belli (2006) generates minimal test sequences, it increases test efficiency both during implementation and execution. Another benefit of this approach for test input generation is it creates a one-to-one mapping between each functional scenario in the form of sequence diagrams and test inputs. A short, independent, and highly cohesive test input is generated for each functional scenario. If one of the functional scenarios changes, this approach ensures only a single test input needs to be updated to keep the test suite up to date, increasing its maintainability.

TOOL SUPPORT

Two tools are implemented for the proposed sequence diagram to ESG transformation and the coverage level measurement of tests on test requirements driven from sequence diagrams.

Transformation rules are implemented using Java and named *SequenceDiagram2ESG* (2023). SequenceDiagram2ESG takes a sequence diagram written in the PlantUML syntax as input and parses it into the simplified sequence diagram model given in Figure 2. Then the sequence diagram is transformed into its equivalent ESG by the transformation rules of the proposed method. The PlantUML syntax is chosen due to it being widely in use in the industry. Other forms of sequence diagrams can be adapted to the tool by rewriting them in the PlantUML syntax, as it imposes no limits on the sequence diagram specifications given in Unified Modeling Language Reference Manual (OMG Unified Modeling Language, 2017). Similarly, the presented tool can also be extended to recognize other sequence diagram forms and syntaxes without further changes to rest of its functionality.

A coverage level measurement tool is implemented in Java and is named *ESGCoverageMeasurer* (2023). As input, ESGCoverageMeasurer takes a set of ESGs and a set of event lists, defining walks over given ESGs. It then calculates the coverage level of these walks for given ESGs and for any length path coverage criteria. It is possible to calculate the functional coverage levels by feeding the ESGCoverageMeasurer the ESGs derived by SequenceDiagram2ESG and the set of method traces from each test run of the software under consideration. To address sequence diagram and object-oriented programming-related issues, ESGCoverageMeasurer accepts two optional inputs. Sequence diagrams may have references to abstract classes or methods. To relate these abstract calls to their concrete implementations in the trace, the tool has an optional input of a class hierarchy forest, depicting the inheritances of interest. If the sequence diagram uses object instance names rather than class names as actors, a second optional input can be provided to relate instance names to class names.

In the event the messages in the sequence diagram denote actual method names within the source code, ESGCoverageMeasurer can automatically match method calls in the test run's execution trace with sequence diagram messages, requiring no change in either sequence diagram or source code. On the other hand, if the sequence diagram's messages do not denote method calls and instead denote a system state or event, it is not possible to directly relate the sequence diagram to the test run trace. In that case, either the sequence diagram should be modified to include method names, or the source code should be modified to output print statements at execution points relevant to the system states or events mentioned in the sequence diagram.

EVALUATION

To evaluate the feasibility of automatically generating ESGs from sequence diagrams, and then use the generated ESGs as a basis to derive test sequences, the SequenceDiagram2ESG tool has been validated against 84 publicly available code repositories. The repositories contain sequence diagrams written in PlantUML (PlantUML Language Reference Guide, 2023) syntax. The sequence diagrams are fed into the SequenceDiagram2ESG individually in order to transform them into ESGs. Then the resulting ESGs are given as an input to the esg-engine tool (2023), which produces minimal test sequences satisfying arbitrary-length path-coverage criteria. The results are summarized in Table 1, and a detailed version is presented in Appendix A. The results show that the average repository contains more than 10 sequence diagrams and the test sequences to provide vertex-pair coverage would contain upwards of 200 test statements. The average number of test statements required for 2- and 3-length-path coverage further increase to approximately 256 and 372, respectively. Without the tool's assistance, deriving these test statements would have required tedious work with sequence diagrams. This manual derivation is also prone to producing less efficient test sequences with a higher number of test statements, as it is **not a trivial** process for larger sequence diagrams. The generality of the proposed transformation method and test sequence generation method is demonstrated by applying the proposed method on 84 arbitrary and public repositories. Their application required no operator involvement or prerequisite on sequence diagrams, other than they be written in PlantUML syntax.

Table 1. Results summary for test sequence generation from sequence diagrams

Number of tested repositories	84.00
Total number of sequence diagrams	871.00
Average number of sequence diagrams per repository	10.37
Average number of generated test statements for vertex-pair coverage	206.23
Average number of generated test statements for 2-length-path coverage	256.61
Average number of generated test statements for 3-length-path coverage	372.40

Averages are provided on a per-repository basis.

The tool can be adapted to other available forms of sequence diagrams. The PlantUML syntax was chosen due to its wide acceptance and abundance of projects using it.

To evaluate the proposed method holistically and to compare generated test sequences with ones that are created manually, an existing public repository is used (The World, n.d.). The repository contains a text-based, multiplayer, and turn-based game. The game is written in Java across 34 source files and 28 Java classes with 1918 lines of code and 211 methods. The repository also contains 23 sequence diagrams written in PlantUML (PlantUML Language Reference Guide, 2023) syntax, depicting various interactions in the game. Some sequence diagrams contain simple interactions within a single frame, while others describe more complex interactions spanning several nested frames. Finally, the repository contains 13 separate acceptance tests in the form of input files to be fed to the compiled program.

A concrete example of how the proposed method works is depicted in Figure 8. It depicts a single sequence diagram and a single set of test inputs from the online project and follows the proposed method's steps. The sequence diagram given in Figure 8a is taken from the online project. It depicts, in a high level of abstraction, how each round of the game will work: each *Virologist* (i.e., the player) instance will be given their turn, orchestrated by a *Controller* instance. The sequence diagram, by the SequenceDiagram2ESG tool, is transformed into an ESG, which is given in Figure 8b.

To measure the functional coverage level of the tests, the ESGCoverageMeasurer tool requires the transformed ESGs, along with test run traces. In order to obtain these traces, the software is executed with the test inputs. The tool then compares the test requirements for a given coverage criterion with the run trace to calculate the coverage level. As an example, for the ESG given in Figure 8b, the vertex-pair coverage criterion results in the following set of test requirements with 5 vertex-pairs:

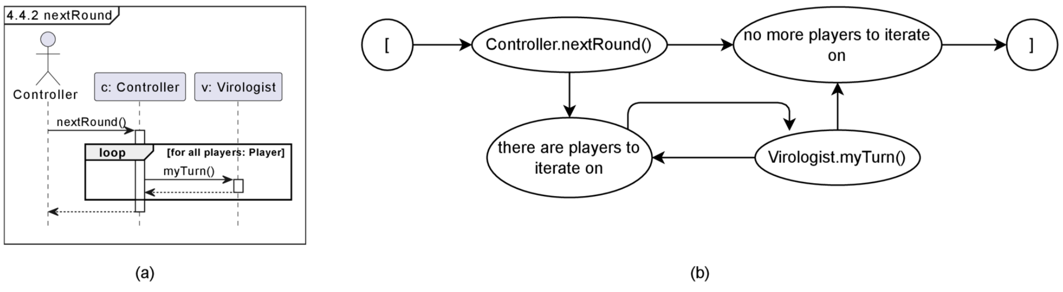
$$TR_{event\_pair} = \{[, \text{Controller.nextRound}(\ ), \text{Controller.nextRound}(\ ),], \text{Controller.nextRound}(\ ), \text{Virologist.myTurn}(\ ), \text{Virologist.myTurn}(\ ), \text{Virologist.myTurn}(\ ), \text{Virologist.myTurn}(\ ),]\}$$

Assuming the run trace obtained from the executed test is as the following:

$$RunTrace = \begin{bmatrix} , \text{Controller.nextRound}(\ ), \text{Virologist.myTurn}(\ ), \ldots, \\ \text{Controller.nextRound}(\ ), \text{Virologist.myTurn}(\ ), \ldots, \end{bmatrix}$$

where the ellipsis represents the occurrence of irrelevant events, functional coverage for the vertex-pair coverage criterion is measured as 3/5 or 60%, since the trace contains 3 of the 5 elements of the test requirements set.

Figure 8. (a) Sequence diagram depicting the actions during a flow and (b) its ESG equivalent



For the ESG given in Figure 8b and coverage criterion of vertex-pair, the esg-engine produces two complete event sequences:

$$CES_1 = \left[,\mathrm{Controller.nextRound}\left(\ \right),\right]$$

and:

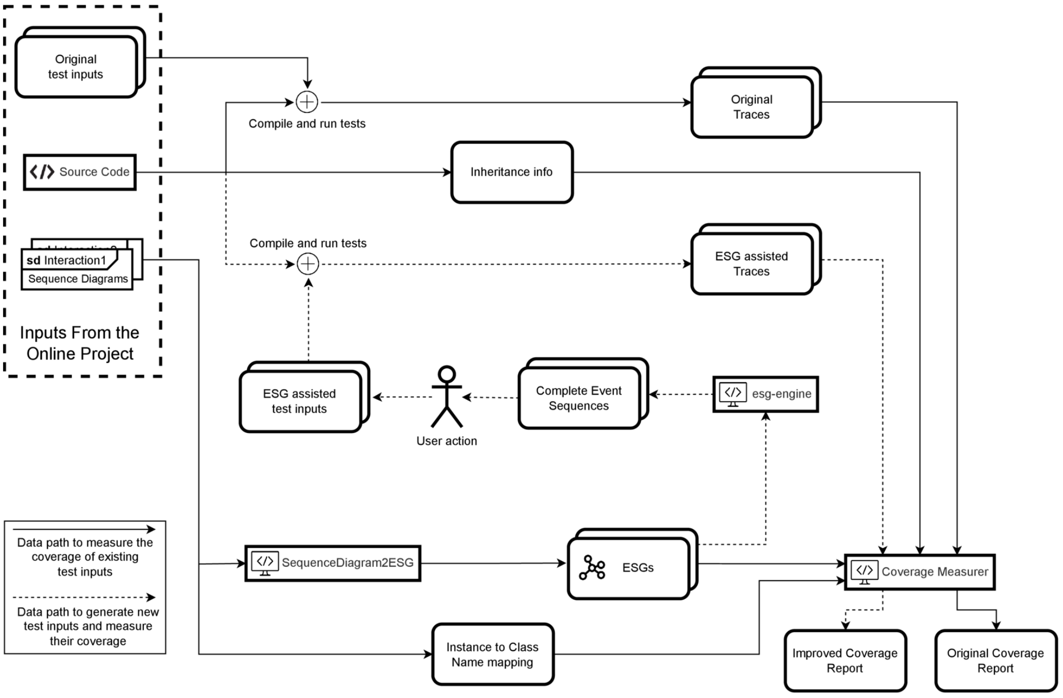
$$CES_2 = \left[,\mathrm{Controller.nextRound}\left(\right),\mathrm{Virologist.myTurn}\left(\ \right),\mathrm{Virologist.myTurn}\left(\ \right),\right]]$$

A set of test inputs can be created to trigger the test sequence CES_2. However, due to the game's rules, the event sequence of "*Controller.nextRound(),]*" is infeasible since it is impossible to play the game with no players at the beginning of a turn. This case represents one reason why transformed ESGs may contain infeasible event sequences. The iteration over the players is represented with a loop frame, which permits skipping the frame's contents. However, the business logic does not permit that.

Finally, the software can be executed with the new set of test inputs, and the resulting trace can be used with the ESGCoverageMeasurer to calculate its coverage level to be 4/5 or 80%. In this case, 80% coverage is the highest possible coverage level due to one of the five elements in the test requirements set being infeasible.

Figure 9 depicts the data flow for the process described above. Three artifacts are taken from the project: source code, set of original test inputs, and sequence diagrams. The source code is compiled and run against the original set of test inputs to produce a set of test run traces. Sequence diagrams are converted into ESGs by the SequenceDiagram2ESG tool. The run traces and ESGs are fed to

Figure 9. Data flow for the proposed method



ESGCoverageMeasurer, along with inheritance info and object instance to class name mappings to calculate the coverage levels of test inputs on sequence diagrams. The same set of ESGs is used with the esg-engine tool (2023) to produce a set of complete event sequences. The event sequences are used as a guide to create a set of test inputs manually. The new set of test inputs is also fed to the ESGCoverageMeasurer, to obtain the coverage levels for the proposed method.

By applying the above-described method to the entire project, 23 ESGs are obtained by the proposed sequence diagram transformation via the SequenceDiagram2ESG. The 13 tests are executed to obtain 13 execution traces. The 23 ESGs and 13 traces from the 13 tests are given to the ESGCoverageMeasurer tool for calculating the coverage levels. Coverage levels of existing test inputs are presented in Table 2 for Vertex coverage, Vertex-pair coverage (1-Length path coverage), and 2-Length path coverage.

Original test inputs hit 95 methods of the 108 methods mentioned in the sequence diagrams. Even though this corresponds to an 88% vertex coverage, it is observed that the missed methods are infeasible – either due to their unreachability or due to sequence diagrams referencing non-existing methods. The coverage level further decreases as the criteria considers longer event tuples. The decrease in coverage level is observed to be mainly caused by additional infeasible tuples.

Due to these infeasible members of the test requirements set, it is not possible to assess the effectivity of the original test inputs in isolation since it is not possible to know the coverage level on the set of feasible test requirements. A meaningful comparison can only be made among different test inputs for the same system.

A second test input set is created by the proposed method: for each sequence diagram, an equivalent ESG is generated by the SequenceDiagram2ESG tool. Then test sequences for 2-length pair coverage are created by the esg-engine tool. Infeasible tuples in the test sequences are detected and removed manually. A test input is manually created for each of the cleaned test sequences. This process ensures that the resulting test inputs will have the highest possible coverage levels, as they satisfy all feasible test requirements. The result is 23 independent test inputs, one for each sequence diagram. Coverage levels of the test inputs created with the proposed method are given in Table 2.

Comparing the original and proposed test inputs, it is observed that the original test inputs achieved the highest possible vertex coverage. For the vertex-pair and 2-length path coverage, original test inputs only miss a single feasible test requirement. However, it is observed that this single missed test requirement can reveal a fault that can be missed even with perfect vertex coverage. As for efficiency, the test inputs generated with the proposed method can achieve the same coverage level as the original test inputs with less than half the number of test statements.

For a project to be a fit for the first part of the evaluation, namely the measurement of coverage level of existing tests, it should have sequence diagrams written with PlantUML syntax and test input sets for acceptance tests. To be able to automatically associate test traces with the messages in the sequence diagram, the messages must be named the same as the method names in the code. Otherwise, a mapping is needed between messages in the sequence diagram and the methods in the

Table 2. Properties of existing test inputs and the test inputs generated by the proposed method

	Original Test Inputs	**Proposed Test Inputs**
Number of Test Runs	13	23
Number of Test Statements	211	97
Vertex Coverage	95/108 (88.0%)	789 (88.0%)
Vertex-pair Coverage	90/125 (72.0%)	123 (72.8%)
2-Length Path Coverage	75/145 (51.7%)	456 (52.4%)
Detected Faults	0	1

code, which requires expert knowledge on the project. For the second part of the evaluation, the test inputs to trigger the complete event sequences generated by the proposed method are to be derived. This, again, requires expert knowledge of the project. To find eligible projects, public repositories on Github are first automatically filtered by existence of sequence diagrams written with PlantUML syntax. Then they are manually filtered by checking whether the messages in sequence diagrams match method names. Finally, they are filtered by checking if they contain executable acceptance tests. Due to these constraints being satisfied only on a small number of projects and the amount of effort required to study the project to be able to create a novel set of test inputs with the proposed method, the method is not evaluated on a second project.

THREATS TO VALIDITY

The main threat to this study's external validity is that its evaluation is done on a single project. Evaluation requires a project with existing sequence diagrams and a test suite. Further searches for a public project satisfying these constraints were unfruitful. Another limiting factor for evaluating more projects is the effort needed to generate test inputs for the test sequences produced by the proposed method. The project must be studied extensively to understand the scenarios that can trigger any given test sequence, along with the deduction of infeasible paths. Nonetheless, it should be noted that the methods proposed in this work are generalizable and applicable to any UML sequence diagram conforming to UML standards. Another threat to this work's validity is the three custom tools (SequenceDiagram2ESG, 2023; ESGCoverageMeasurer, 2023; esg-engine, 2023) that are implemented by the research group. Although they have been tested extensively without revealing any faults, any underlying defects may have an impact on the results presented in this work.

CONCLUSION

This study proposes a method to transform UML sequence diagrams into event sequence graphs (ESG). With the proposed method, it is possible to measure the functional coverage levels of existing tests. Transformation to ESG also enables the automatic generation of test sequences for certain coverage criteria and aids in creating effective, efficient, and maintainable tests.

The proposed transformation and test sequence generation method has been validated with 84 public repositories, which showed that it is applicable to a wide range of projects without any constraints, and test sequences can be generated without changes to existing sequence diagrams and without user involvement. The efficiency and effectiveness of the test sequences generated by the proposed method is evaluated on a public repository. It is shown that the manual generation of test inputs may lead to reduced coverage levels for more complex coverage criteria, and this may result in not detecting some faults that should have been revealed with given coverage criteria. It is also shown that test inputs generated with the proposed methods are more efficient. A higher coverage level than the original test inputs with 211 statements is achieved with the proposed method with only 97 statements. Even though the number of test runs is increased from 13 to 23 with the proposed method, each run contains fewer statements because they address a single use case described with a single sequence diagram. The evaluation could not be extended to more projects due to its requiring projects to contain sequence diagrams and a test suite.

REFERENCES

Alferez, M., Pastore, F., Sabetzadeh, M., Briand, L., & Riccardi, J. R. (2019). *Bridging the gap between requirements modeling and behavior-driven development.* [Presentation]. The 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), Munich, Germany.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press. doi:10.1017/9781316771273

Atem de Carvalho, R., Soares Manhães, R., & Others. (2010). *Filling the gap between business process modeling and behavior driven development*. ArXiv E-Prints, arXiv-1005.

Belli, F. (2001, Month Day). *Finite state testing and analysis of graphical user interfaces.* [Proceedings] The 12th International Symposium on Software Reliability Engineering, Hong Kong.

Belli, F., Beyazit, M., & Memon, A. (2012, June 20). Testing is an event-centric activity. *The 2012 IEEE Sixth International Conference on Software Security and Reliability.* IEEE. doi:10.1109/SERE-C.2012.24

Belli, F., & Budnik, C. J. (2007). Test minimization for human-computer interaction. *Applied Intelligence*, *26*(2), 161–174. doi:10.1007/s10489-006-0008-0

Belli, F., Budnik, C. J., & White, L. (2006). Event-based modelling, analysis and testing of user interactions: Approach and case study. *Software Testing, Verification & Reliability*, *16*(1), 3–32. doi:10.1002/stvr.335

Belli, F., Nissanke, N., Budnik, C. J., & Mathur, A. (2005). Test generation using event sequence graphs. *Software Engineering*, 52.

Bernardi, S., Donatelli, S., & Merseguer, J. (2002, July 24). *From UML sequence diagrams and state charts to analysable petri net models.* [Proceedings] The 3rd International Workshop on Software and Performance, Rome, Italy.

Cartaxo, E. G., Neto, F. G., & Machado, P. D. (2007, October). *Test case generation by means of UML sequence diagrams and labeled transition systems.* The 2007 IEEE International Conference on Systems, Man and Cybernetics, Montreal, Quebec, Canada. doi:10.1109/ICSMC.2007.4414060

Chandnani, K., Patidar, C. P., & Sharma, M. (2017). Automatic and optimized test case generation using model based testing based on sequence diagram and discrete particle swarm optimization algorithm. [IRJET]. *International Refereed Journal of Engineering & Technology*, *1*(2), 1–6.

Cockburn, A. (1998). Basic use case template. *Humans and Technology*. *Technical Report*, *96*, 28.

de Carvalho, R. A., Manhaes, R. S., & Others. (2010). *Mapping business process modeling constructs to behavior driven development ubiquitous language.* ArXiv Preprint ArXiv:1006. 4892.

Dijkman, R. M., Dumas, M., & Ouyang, C. (2008). Semantics and analysis of business process models in BPMN. *Information and Software Technology*, *50*(12), 1281–1294. doi:10.1016/j.infsof.2008.02.006

ESG Coverage Measurer public repository. (2023). Github. https://github.com/esg4aspl/esg-coverage-measurer

esg-engine public repository. (2023). Github. https://github.com/esg4aspl/esg-engine

Gherkin reference. (n.d.). Cucumber. https://cucumber.io/docs/gherkin/reference/

Gutiérrez, J. J., Ramos, I., Mejías, M., Arévalo, C., Sánchez-Begines, J. M., & Lizcano, D. (2017, September). *Modelling Gherkin Scenarios Using UML.* The 26th International Conference on Information Systems Development. Pyla, Larnaca, Cyprus.

Hoseini, B., & Jalili, S. (2014, September). *Automatic test path generation from sequence diagram using genetic algorithm.* The 7th International Symposium on Telecommunications, Tehran, Iran. doi:10.1109/ISTEL.2014.7000678

Jena, A. K., Swain, S. K., & Mohapatra, D. P. (2015). Test case creation from UML sequence diagram: A soft computing approach. *Intelligent Computing, Communication and Devices: Proceedings of ICCD 2014, 1*, (pp. 117-126). IEEE.

Khandai, M., Acharya, A. A., & Mohapatra, D. P. (2011, April). *A novel approach of test case generation for concurrent systems using UML sequence diagram.* The 3rd International Conference on Electronics Computer Technology, Kanyakumari, India. doi:10.1109/ICECTECH.2011.5941581

Knuth, D. E., Morris, J. H. Jr, & Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing*, *6*(2), 323–350. doi:10.1137/0206024

Lei, Y. C., & Lin, N. W. (2008). *Semiautomatic Test Case Generation Based on Sequence Diagrams*. Springer.

Li, B. L., Li, Z. S., Qing, L., & Chen, Y. H. (2007, December). *Test case automate generation from UML sequence diagram and OCL expression.* The 2007 International Conference on Computational Intelligence and Security, Harbin, China. doi:10.1109/CIS.2007.150

Li, X., Liu, Z., & Jifeng, H. (2004). *A formal semantics of UML sequence diagram*. [Proceedings]. The 2004 Australian Software Engineering Conference, Melbourne, Victoria, Australia. doi:10.1109/ASWEC.2004.1290469

Lübke, D. (2006). *Transformation of Use Cases to EPC Models*. Citeseer.

Lubke, D., Schneider, K., & Weidlich, M. (2008). *Visualizing use case sets as BPMN processes.* 2008 Requirements Engineering Visualization, Barcelona, Spain. 10.1109/REV.2008.8

Mallick, A., Panda, N., & Acharya, A. A. (2014). Generation of test cases from uml sequence diagram and detecting deadlocks using loop detection algorithm. *International Journal on Computer Science and Engineering*, *2*, 199–203.

OMG® Unified Modeling Language® Version 1.5. (2003). Object Management Group. https://www.omg.org/spec/UML/1.5

OMG® Unified Modeling Language® Version 2.5. (2015). Object Management Group. https://www.omg.org/spec/UML/2.5

Priya, S. S., & Malarchelvi, P. S. K. (2013). Test path generation using UML sequence diagram. *International Journal of Advanced Research in Computer Science and Software Engineering*, *3*(4).

Roques, A. (n.d.). *Plantuml language reference guide*. PlantUML. https://plantuml.com/guide

Sarma, M., Kundu, D., & Mall, R. (2007, December). *Automatic test case generation from UML sequence diagram*. The 15th International Conference on Advanced Computing and Communications, Guwahati, India doi:10.1109/ADCOM.2007.68

SequenceDiagram2ESG public repository. (2023). Github. https://github.com/esg4aspl/sequence-diagram-to-esg

Sumalatha, V. M. (2013). Object oriented test case generation technique using genetic algorithms. *International Journal of Computer Applications*, *61*(20).

Swain, S. K., Mohapatra, D. P., & Mall, R. (2010). Test case generation based on use case and sequence diagram. *International Journal of Software Engineering*, *3*(2), 21–52.

The world of blind virologists. (n.d.). Github. https://github.com/Tschonti/503-service-unavailable

Van der Aalst, W. M. P. (1999). Formalization and verification of event-driven process chains. *Information and Software Technology*, *41*(10), 639–650. doi:10.1016/S0950-5849(99)00016-6

Zhang, C., Duan, Z., Yu, B., Tian, C., & Ding, M. (2016). A test case generation approach based on sequence diagram and automata models. *Chinese Journal of Electronics*, *25*(2), 234–240. doi:10.1049/cje.2016.03.007

APPENDIX

Table 3. Detailed results for sequencediagram2esg evaluation on public repositories

Repository URL	Number of Sequence Diagrams	Vertex-pair Coverage		2-length-path Coverage		3-length-path Coverage	
		Number of Test Runs	Number of Test Statements	Number of Test Runs	Number of Test Statements	Number of Test Runs	Number of Test Statements
https://github.com/tiagoavcosta25/LAPR4	133	182	1672	182	1672	213	2242
https://github.com/telstra/open-kilda	69	123	1914	175	2765	265	4314
https://github.com/Sigmoidal8/Projeto-2-Ano-ISEP-2020-2021	61	84	1416	86	1452	95	1593
https://github.com/Tschonti/503-service-unavailable	50	88	518	113	728	166	1225
https://github.com/0chain/gosdk	44	48	436	50	456	54	528
https://github.com/Flank/flank-dashboard	39	53	629	58	675	64	769
https://github.com/IHE/sdpi-fhir	38	68	484	68	520	68	567
https://github.com/eclipse-tractusx/item-relationship-service	28	43	509	44	559	52	783
https://github.com/mstable/mStable-process-docs	28	39	860	42	966	47	1093
https://github.com/SODALITE-EU/project-wide-documentation	24	47	852	55	1035	81	1677
https://github.com/catenax-ng/product-item-relationship-service	18	25	299	27	348	32	452
https://github.com/Click2Cloud/mizar	16	16	99	16	99	16	99
https://github.com/eclipse-dataspaceconnector/DataSpaceConnector	15	19	180	24	249	30	342
https://github.com/DavidHartman-Personal/PlantUML-Reference	14	16	162	20	306	20	306
https://github.com/Checkmk/checkmk	13	13	170	13	170	13	170
https://github.com/wazuh/wazuh	13	33	360	38	448	45	614
https://github.com/nonodev96/THUMDER	12	12	94	12	94	12	94
https://github.com/nosqlbench/nosqlbench	12	15	199	15	199	18	275
https://github.com/nickfox-taterli/arm-trusted-firmware	10	11	169	11	169	11	169
https://github.com/ostelco/ostelco-core	10	11	111	11	111	11	111
https://github.com/aak74/kubernetes-for-beginners	9	9	52	9	52	9	52
https://github.com/nemerosa/ontrack	8	8	78	8	78	8	78
https://github.com/screwdriver-cd/screwdriver	8	9	64	9	64	9	64
https://github.com/1190452/LAPR2	7	10	173	12	223	14	295
https://github.com/jet/equinox	7	14	406	16	461	16	461
https://github.com/Agoric/agoric-sdk	6	6	107	6	107	6	107
https://github.com/alexa/alexa-auto-sdk	6	8	30	6	30	6	30
https://github.com/AYCH-Inc/aych.hyper.tolerant	6	10	474	10	474	10	474
https://github.com/d3sw/conductor	6	9	176	8	175	8	175
https://github.com/internetarchive/iari	6	9	147	12	256	13	314

continued on following page

Table 3. Continued

Repository URL	Number of Sequence Diagrams	Vertex-pair Coverage		2-length-path Coverage		3-length-path Coverage	
		Number of Test Runs	Number of Test Statements	Number of Test Runs	Number of Test Statements	Number of Test Runs	Number of Test Statements
https://github.com/ProzorroUKR/openprocurement.api	6	9	84	14	117	32	234
https://github.com/RoboCup-SSL/ssl-game-controller	6	11	136	10	136	14	190
https://github.com/skhoroshavin/indy-plenum	6	10	474	10	474	10	474
https://github.com/CaliOpen/Caliopen	5	9	159	10	174	11	191
https://github.com/glide-im/glide-im	5	12	65	12	65	12	75
https://github.com/golemfactory/yapapi	5	8	99	9	120	10	144
https://github.com/jgomezselles/hermes	5	6	35	6	35	6	35
https://github.com/ntan1902/hey-app	5	6	43	6	43	6	43
https://github.com/ThoughtsBeta/flash-sale	5	5	58	15	226	15	226
https://github.com/Akayeshmantha/indy-plenum	4	5	208	5	208	5	208
https://github.com/CorfuDB/CorfuDB	4	5	61	5	61	5	61
https://github.com/gematik/api-erp	4	5	65	5	65	6	81
https://github.com/hyperledger/fabric-private-chaincode	4	4	21	4	21	4	21
https://github.com/jamro/jsbattle	4	4	45	4	45	4	45
https://github.com/JCSDA-internal/ioda-converters	4	8	193	12	312	15	478
https://github.com/MyPureCloud/genesys-messenger-transport-mobile-sdk	4	7	130	7	130	7	130
https://github.com/nhsconnect/integration-adaptor-gp2gp	4	4	47	4	47	4	47
https://github.com/republique-et-canton-de-geneve/chvote-protocol-poc	4	7	56	6	56	7	68
https://github.com/TresAmigosSD/SMV	4	4	82	4	82	4	82
https://github.com/UWB-Biocomputing/Graphitti	4	6	157	8	226	8	226
https://github.com/VForWaTer/vforwater-portal	4	11	711	21	2208	32	6606
https://github.com/aim42/htmlSanityCheck	3	8	58	13	102	21	181
https://github.com/akvo/akvo-rsr	3	4	60	4	60	5	83
https://github.com/aws-samples/sagemaker-ssh-helper	3	3	123	3	123	3	123
https://github.com/if-h4102/pld-agile	3	4	84	4	116	3	180
https://github.com/opensearch-project/OpenSearch-Dashboards	3	3	27	3	27	3	27
https://github.com/umati/TransformationEngineApi	3	3	18	3	18	3	18
https://github.com/uniba-dsg/betsy	3	3	20	3	20	3	20
https://github.com/cloudfoundry-community/gautocloud	2	5	35	5	35	5	35
https://github.com/fp7-netide/Engine	2	3	53	3	53	4	66
https://github.com/hashgraph/hedera-mirror-node	2	2	28	2	28	2	28

continued on following page

Table 3. Continued

Repository URL	Number of Sequence Diagrams	Vertex-pair Coverage		2-length-path Coverage		3-length-path Coverage	
		Number of Test Runs	Number of Test Statements	Number of Test Runs	Number of Test Statements	Number of Test Runs	Number of Test Statements
https://github.com/kaltura/player-sdk-native-android	2	2	25	2	25	2	25
https://github.com/OpenFunction/functions-framework-python-gcp	2	3	51	3	51	3	51
https://github.com/sagikazarmark/temporal-intro-workshop	2	3	22	2	22	2	22
https://github.com/Telenav/kivakit	2	3	15	3	15	3	15
https://github.com/tlodderstedt/openid-connect-4-credential-issuance	2	2	17	3	27	3	27
https://github.com/v8platform/protos	2	3	56	7	142	13	294
https://github.com/videojs/http-streaming	2	4	77	4	77	5	109
https://github.com/xuanye/plantuml-style-c4	2	4	19	5	25	6	32
https://github.com/ycc140/fastapi_messaging	2	2	97	2	97	2	97
https://github.com/asynkron/protoactor-go	1	2	12	2	12	2	12
https://github.com/baloo/cargo-readme	1	2	56	2	56	2	56
https://github.com/Bannerlord-Coop-Team/BannerlordCoop	1	3	32	3	39	3	39
https://github.com/Cerulean-Circle-GmbH/once.sh	1	1	327	1	327	1	327
https://github.com/eyebluecn/smart-classroom-misc	1	2	41	2	41	2	41
https://github.com/Jason7jl/SustainabilityShipmentService	1	2	22	4	39	4	41
https://github.com/LemADEC/WarpDrive	1	2	16	3	23	4	32
https://github.com/mimblewimble/grin-wallet	1	1	17	1	17	1	17
https://github.com/PelionIoT/java-coap	1	2	32	2	32	2	32
https://github.com/SiliconLabs/UnifySDK	1	1	12	1	12	1	12
https://github.com/SvenskaSpel/locust-swarm	1	1	6	1	6	1	6
https://github.com/taosdata/taosadapter	1	2	78	2	78	2	78
https://github.com/w3c/webpayments-flows	1	1	10	1	10	1	10
https://github.com/xnuter/http-tunnel	1	1	8	1	8	1	8

Nazım Umut Ekici received his MS in Electrical and Electronics Engineering from the Middle East Technical University (METU) in 2019. He is currently a third-year Ph.D. student in Computer Engineering at Izmir Institute of Technology (IZTECH). Alongside his academic studies, he has been working as a software engineer since 2016. His interests are software reliability, validation, and software specification languages.

Tugkan Tuglular received the B.S., M.S., and Ph.D. degrees in Computer Engineering from Ege University, Turkey, in 1993, 1995, and 1999. He worked as a research associate at Purdue University from 1996 to 1998. He has been with Izmir Institute of Technology since 2000. After becoming an Assistant Professor at Izmir Institute of Technology, he worked as Chief Information Officer in the university from 2003-2007. In addition to his academic duties, he acted as IT advisor to the Rector between 2010-2014. In 2018, he became an Associate Professor in the Department of Computer Engineering of the same university. He has more than 75 publications and an active record of duties with international and national conferences. His current research interests include model-based testing and software quality.