IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 10, 2020, accepted March 11, 2020, date of publication March 18, 2020, date of current version April 15, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2981742

Improving the Security of UML Sequence Diagram Using Genetic Algorithm

MOHAMMAD ALSHAYEB, HARIS MUMTAZ, SAJJAD MAHMOOD, AND MAHMOOD NIAZI
Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia
Electrical, Computer and Software Engineering Department, The University of Auckland, Auckland 1142, New Zealand

Corresponding author: Sajjad Mahmood (smahmood@kfupm.edu.sa)

This work was supported by the Deanship of Scientific Research at King Fahd University of Petroleum and Minerals, Saudi Arabia.

ABSTRACT A sequence diagram is a modeling approach for visualizing the behavioral execution of a system. The objective of this research is to investigate the problem of security in a behavioral model (sequence diagram) through the application of model refactoring. We propose detection and correction techniques, empirical evaluation of the proposed techniques and assessment of security improvement in sequence diagrams. The detection of security bad smells is achieved through the adaptation of a genetic algorithm, while correction is accomplished by the model transformation approach. The results show significant detection recall and correction efficacy of the proposed detection and correction approaches, respectively. Our results show that the proposed approach is effective in detecting and correcting bad smells and can improve the security of UML Sequence Diagram.

INDEX TERMS Software security, security bad smells, software refactoring, genetic algorithm, software metrics.

I. INTRODUCTION

The Unified Modeling Language (UML) is a widely used analysis and design language due to its support for a number of software quality attributes [2]. It allows the designers to develop analysis and design models ensuring important quality attributes. A behavioral view of UML provides a graphical visualization of the behavioral execution of a system. A sequence diagram is a popular technique to visually represent the behavioral dynamics of a system in the forms of lifelines and their interactions. Since a sequence diagram is developed at the design phase, any bad smells in a sequence diagram can easily propagate to subsequent software artifacts. Therefore, it becomes crucial to ensure quality in sequence diagrams. Refactoring is a technique that improves the quality of a software artifact without altering its behavior [3]. The correction of bad smells in sequence diagrams is extremely important to enhance the quality of this behavioral model.

A number of quality attributes related to software modeling have been reported in the literature, such as modularity, reusability, modifiability, testability, security etc. [4], [5]. These days, security has become one of the important quality attributes in software systems because of the critical nature of systems. This is also evident from the abundance of literature on secure software development in recent years. However, there is a lack of work on investigating bad smells and their impact on the quality of sequence diagrams. In addition, there is also a scarcity of work on evaluating the impact of refactoring on improving the quality of sequence diagrams [6]. Designers and developers should be careful when refactoring security critical code as it can introduce subtle leaks [7]. This work fills these gaps by providing detection and correction approaches to respectively identify and remove security bad smells in sequence diagrams. The work presented in this paper was conducted as part of an MSc thesis [8].

The main goal of this research is to improve the security of sequence diagrams through the application of refactoring. The achievement of this goal can be broken down into multiple sub-objectives. The sub-objectives of this research include:

- to propose a detection technique to identify security bad smells in sequence diagrams.
- to propose a correction technique to eradicate security bad smells in sequence diagrams.
- to empirically assess security improvements in sequence diagrams due to refactoring.

The associate editor coordinating the review of this manuscript and approving it for publication was Resul Das.

This research addresses the following research questions:

RQ1: To what extent can the proposed detection approach detect security bad smells in sequence diagrams?

RQ2: To what extent can the proposed correction approach rectify security bad smells in sequence diagrams?

RQ3: To what extent can refactoring security bad smells improve the security aspects of sequence diagrams?

To address RQ1 and RQ2, we propose detection and correction approaches, respectively. The detection approach uses the concept of Genetic Algorithms (GAs) to identify security bad smells in the studied sequence diagrams. A potential solution is formed by creating a set of rules measuring security bad smells using quality metrics. The best solution is yielded through the selection, crossover and mutation operations of the GA process. Initially, we build the detection rules from five sequence diagrams, but to justify the use of GA, we generated detection rules for a large dataset of sequence diagrams. The correction solution is based on model transformation using XMI. The XML representation of a corresponding model is refactored to remove security bad smells. The refactored XML is then exported to the corresponding sequence diagram. RQ3 is answered through the analysis of sequence diagram case studies and the statistical analysis of quality metrics. The comparison of software metric values before and after refactoring allows a definite conclusion on significant security improvement in sequence diagrams.

The rest of this paper is structured as follows: section 2 provides a detailed description of the related work. Section 3 presents the research methodology, including details of the proposed detection and correction approaches. Section 4 discusses the implementation details of the proposed detection and correction approaches on multiple case studies of sequence diagrams. It also describes the validation of the proposed approaches through case studies and the statistical analysis of quality metrics. Section 5 presents the analysis and a discussion of the implications of the acquired results. Section 6 presents the threats to validity and finally, Section 7 concludes the paper and directs future work.

II. RELATED WORK

This section provides a detailed discussion on studies related to this research. It covers model bad smells and existing bad smell detection techniques.

A. MODEL BAD SMELLS

Suryanarayana *et al.* classified design smells into four categories: abstraction, encapsulation, modularization and hierarchy [9]. The classification with its corresponding design smells is listed in Table 1. In each classification, a significant number of design smells are reported. For example, in abstraction, there is a design smell, ''missing abstraction'', which emphasizes a compromise on the integrity of data. Similarly, in deficient encapsulation, the class attributes are likely to be exposed to outside classes. The authors also suggested an appropriate set of refactoring opportunities for each design smell and their impact on quality attributes.

TABLE 1. Classification of design smells [9].

Classification	Design Smells
Abstraction	Missing, Imperative, Incomplete, Multifaceted, Unnecessary, Unutilized, Duplicate
Encapsulation	Deficient, Leaky, Missing, Unexploited
Modularization	Broken, Insufficient, Cyclically Dependent, Hub-like
Hierarchy	Missing, Unnecessary, Unfactored, Wide, Speculative, Deep, Rebellious, Broken, Multipath, Cyclic

The design bad smells that are considered in this paper are briefly described as in Table 1.

Missing Modularization: This type of design smell arises when a class or component is not decomposed. In other words, the component lacks the separation of concerns. The appropriate refactoring strategy for removing this design smell is 'Extract Class' as the cohesive attributes and methods are moved to a new class, which leaves the old class properly modularized.

Broken Modularization: This is when the data and related procedures are split across abstractions, which allows the unauthorized access of data across classes or components. The related refactoring to eradicate this design smell is move method(s) and attribute(s), which means the data and related procedures are moved to the class(es) where they actually belong.

Unutilized Abstraction: This design smell occurs when an unused abstraction is accidentally invoked, which may result in runtime problems, affecting the reliability of a design. This type of design smell can be erased by the application of remove abstraction refactoring. Removal of unutilized abstraction motivates the correct invocation of objects, which results in the reliable execution of software.

Bad smells in the sequence diagram are studied in a similar manner as class diagram bad smells, the rationale based on the similarities between both models in terms of classes and interactions. Hence, bad smells, which belong to class diagrams, are applicable to a sequence diagram. For example, broken modularization is one of the bad smells usually experienced in a class diagram and it can also be applied to a sequence diagram. The calling of methods between classes identifies how much classes depend on each other intimating the presence of broken modularization. The rectification procedure is also like the class diagram. If the bad smells are removed at the class level, they are automatically removed from the sequence diagram. Following the same broken modularization example, the methods are moved to remove the defects in the classes, meaning less calling of methods, and eventually, fewer calls are observed in corresponding sequence diagrams.

B. BAD SMELL DETECTION TECHNIQUES

The classification which we consider in this research is presented by Misbhauddin and Alshayeb [6]. They classified three detection strategies, namely: design patterns, software metrics and pre-defined rules [6]. The details of the detection techniques in terms of this classification is presented in this section.

Software metrics provide statistical information by capturing its key attributes. The most well-known object-oriented metrics reported in the literature are proposed by Chidamber and Kemerer [10]. Software metrics usually cannot be directly applied to UML models; hence models are first transformed into XML and then the XML representation is parsed to measure the software metrics values. If the measured metric values are not in an acceptable range, it is considered as a bad smell. Hence, the major concern in metrics-based techniques is the threshold values that can be considered acceptable.

Fourati *et al.* proposed an approach to identify anti-patterns at the structural and the behavioral levels through the use of quality metrics [11]. The structural and behavioral level models considered in their study were class diagrams and sequence diagrams, respectively. The basic purpose of incorporating sequence diagrams was to compensate for the loss of information when moving from the source code to the design. The approach involves several steps. First, the relationship between bad smells and metrics is revealed. Detection is done by transforming class diagrams into XML and then the software metrics are used to identify whether diagrams carry bad smells or not. Mohamed *et al.* also presented an extended meta-model of UML to assist model-driven refactoring [12]. Their approach allows the automated detection of bad smells in class and sequence diagrams by the use of model metrics and design smells. They performed domain analysis to propose a UML extended meta-model to achieve their objective. Once the design smells are identified, the proposed meta-model is applied to find possible refactoring. Refactoring tags are assigned to the source model, indicating the need for restructuring. Before appropriate refactoring is applied, the user validates the refactoring tags. Mumtaz *et al.* evaluated how refactoring can be used to improve the software security at code level [13]. Ruland *et al.* proposed a method to maximize strictness of declared accessibility of class members [14].

Design patterns provide good solutions for defects in software development; on the other hand, anti-patterns provide bad solutions to problems. They allow developers to identify common design and implementation problems and provide an appropriate solution. Improving design quality attributes in models by incorporating patterns into a design is called pattern-based model refactoring [15]. The refactoring procedure based on patterns involves three stages: the setting of the source, setting of the target model and applying transformation [15]. The part of the software artifact which needs refactoring is first selected, then based on a design pattern, a target model is set. The selected portion of the artifact is transformed in accordance with the defined target model. Kim defined a design pattern consisting of three components: problem models, solution models and transformation models [16]. The transformation model describes how problem specification can be transformed to a solution specification. A problem specification is assessed against a specific design pattern for its applicability to that problem. If the pattern specification matches the problem specification, the corresponding transformation model is applied. They provided refactoring specifications for the Abstract Factory pattern, Adapter pattern and Observer pattern.

Rule-based techniques ensure the use of a specific template or standard rules to develop software artifacts. If a software artifact is not created using a predefined standard, it is suspected to have bad smells. Dobrzanski and Kuzniarz presented an approach to systematically specify bad smells and the associated refactoring in the sequence diagram [17]. They used a template that includes the following information: name of refactoring, origin, trigger element, goal, reasons, bad smell, pre and post conditions. They considered UML models that are built in the TAU CASE tool. Jensen and Cheng proposed an approach, based on genetic programming, to automate the use of software metrics to generate refactoring strategies that introduce design patterns [18]. Rasool and Arshad conducted a survey on tools and techniques that have been used for mining code smells at software code level [19]. Khelladi *et al.* proposed a detection engine of complex changes that addresses the variability and the overlap challenges [20].

A few observations can be made from the explanations of the presented detection techniques. The detection of bad smells in sequence diagrams is accomplished via design patterns, software metrics and pre-defined rules. Sequence diagrams are studied alongside class diagrams. The reason for this is due to the similar type of model smells for both diagrams and the way they are detected. The literature has not studied bad smells from a security perspective. Researchers have also not yet addressed the detection and refactoring of bad smells from a security perspective.

C. APPLIED REFACTORING STRATEGIES

Many refactoring strategies have been reported in the literature [3], but those applied in this research are briefly explained in this section. The refactoring strategies for class diagrams and sequence diagrams are generally the same. The rationale behind this is the use of classes and their interactions in class and sequence diagrams. The refactoring strategies considered in this research are briefly explained as follows:

Move Method: This means moving a method from a class to another class that uses this method more. This method can be turned into a delegation or can be completely removed from it. The corresponding bad smell for this type of refactoring is ''broken modularization'' i.e. a method which uses more features of another class than the class to which it belongs.

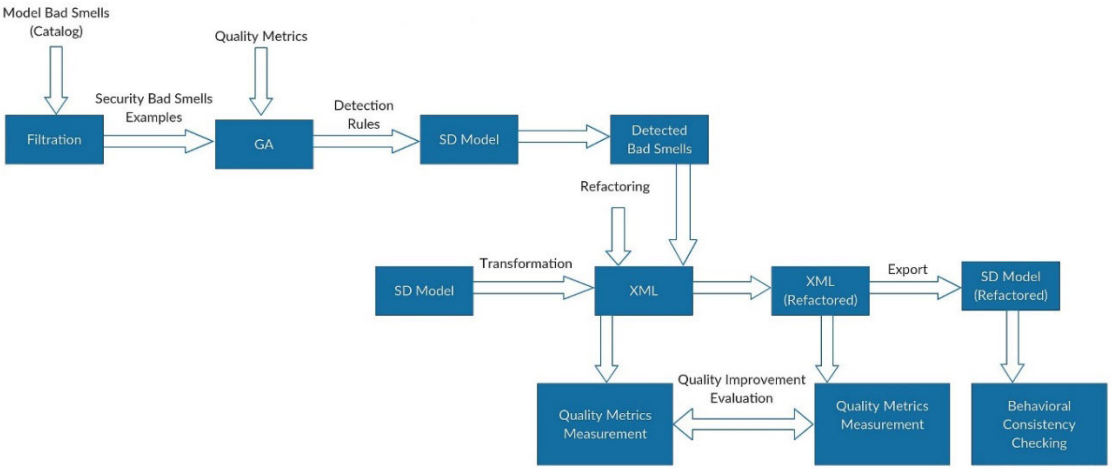

FIGURE 1. Research methodology overview.

This bad smell exploits the common object-oriented design principle of modularization. Although modularization is introduced by distributing the methods across multiple classes, the separation of concerns is not ensured. The move method refactoring allows the method to move to the class where it is most required. This way modularization and the separation of concerns are complementary.

Extract Class: This means moving cohesive methods and related attributes from an existing class to a new class. This type of refactoring handles the proper modularization of design. This type of refactoring copes with the bad smell of missing modularization. This bad smell also violates the separation of concerns principle in software design. In other words, a class is overburdened by many responsibilities. In this way, high coupling is also induced. In order to cope with all these design principle violations, extract class refactoring is applied. The refactoring readjusts the design with the objective of having better modularization and separation of concerns and reduced coupling.

Remove Class: This means removing a class that is not contributing to a design. Such a class becomes useless and it is pointless to show it in a design. The idle class may incur inappropriate behavior in the case where it is accidently invoked. Thus, removing such a class also increases the reliability of a design.

III. RESEARCH METHODOLOGY

This section highlights the major aspects of the research methodology. The main research goal of the methodology is the efficient detection and correction of security bad smells. Another important research goal which needs to be addressed by the research methodology is the evaluation of security improvement because of refactoring security bad smells. Although surveying of security bad smells and quality metrics is performed, the focus of this section is on filtering security bad smells and related refactoring strategies; the detection and correction of security bad smells; and the evaluation of security improvement in the studied sequence diagrams because of refactoring.

A. RESEARCH METHODOLOGY OVERVIEW

This section includes an abstract description of the research methodology, the details of which are presented in the forthcoming section. To ease the understandability, a pictorial view of the research methodology is depicted in Figure 1.

The filtration of security bad smells from the existing taxonomy of bad smells is the initializing activity of the research methodology. A large taxonomy of bad smells exists in the literature, which must undergo some filtration process to strain only security bad smells. After the filtration, security bad smells and the related quality metrics are input to the GA for the generation of detection rules. The rules are combinations of conditional statements using quality metrics. The generated detection rules are then applied on sequence diagrams to detect any bad smells existent in them. This completes the detection activity. The subsequent focus of the research methodology is on the correction of the detected security bad smells. The correction adopts the model transformation approach in which the investigated sequence diagrams are transformed into XML representations. The refactoring strategies are applied to XML representations of the sequence diagrams based on the targeted bad smells. This results in the generation of refactored XML representations. The refactored XML representations are then exported to corresponding refactored sequence diagrams. This completes the correction activity. Behavioral consistency is accomplished using post-refactoring conditions. The quality metrics are computed, before and after refactoring, from XML representations of the sequence diagrams. The improvement in model quality is assessed by performing a comparison of quality metrics computed pre- and post-refactoring. The basic flow of activities is shown in Figure 1.

B. FILTERING OF SECURITY BAD SMELLS

It is important to determine whether bad smells tagged as security bad smells violate one or more security attributes. This process is supported by the study of Suryanarayana *et al.*, in which they classified many design smells [9]. They also identified the quality attributes each design smell could affect. The quality attributes include security related attributes as well. This allows us to identify the model bad smells that violate security attributes and subsequently, tag them as security bad smells. The existing catalog of model bad smells is passed through this process to analyze which model bad smell affects the security attributes. For example, according to Suryanarayana *et al.*, the missing modularization model smell affects understandability, changeability, extensibility, reusability, testability and reliability [9]. According to the security definitions, reliability is one of the security attributes [4], [21]–[23]. Hence, missing modularization can be tagged as a security bad smell. Similarly, other security bad smells are filtered. The forthcoming example identifies a bad smell as a non-security bad smell using the same procedure. According to Suryanarayana *et al.*, imperative abstraction impacts understandability, changeability, extensibility, reusability and testability [9]. Since none of these quality attributes is related to security as per the security definitions, this bad smell is not filtered by the process. In a similar manner, other non-security bad smells are identified. In the scope of this research, we focus on three security bad smells: unutilized abstraction, broken modularization and missing modularization. A brief definition of each security bad smell, along with the security requirements it violates, and appropriate refactoring, are presented in Appendix.

1) APPROACH OVERVIEW

The gaps identified in the literature are addressed by the proposed detection and correction approaches. The idea of the detection approach is inspired by the technique presented by Ouni *et al.* [24], with changes in the GA process, specifically for crossover and mutation operations. In addition, the approaches investigate security bad smells rather than normal class level bad smells. A different set of studied bad smells stipulates the use of a different set of quality metrics. In addition, a major contribution resides in the application of the approach to sequence diagrams.

The security bad smell examples and quality metrics are used to generate detection rules through the application of the genetic algorithm. The detection rules use a set of metrics and their values to detect a specific defect. The set of metrics measuring a bad smell is used as a rule for the detection of that bad smell. The solution generated by the detection approach carries the set of rules, which detects a maximum number of bad smells. The correction approach refactors sequence diagrams using the model transformation technique.

2) GA IMPLEMENTATION FOR THE DETECTION APPROACH

This section demonstrates the application of genetic programming in the context of bad smell detection.

```
Input:
Quality metrics
Security bad smell examples
Process:
1.      I = set of rules
2.      P = set of I
3.      S = Sequence diagram
4.      repeat
5.      for all I in P do
6.              detected bad smells =
execute_rules(S);
7.              fitness (I) =
numberOfDetectedBadSmells;
8.      end for
9.      best_solution =
best_fitness(I);
10.     P = new_population(P);
11.     it = it + 1;
12. until it = max_bad_smells;
13. return best_solution;
Output:
best_solution
```

FIGURE 2. A high-level GA adaptation for detection [24].

R1: IF (NAss(c) >= 38 AND (NInvoc(c) >= 11 AND NRec(c) >= 19) AND CBO(c) >= 3) THEN missing modularization(c)
R2: IF (NAss(c) == 2 AND (NInvoc(c) == 0 OR NRec(c) == 1) AND CBO(c) == 1) THEN broken modularization(c)
R3: IF (NAss(c) == 0 AND NInvoc(c) == 0 AND NRec(c) == 0 AND CBO(c) == 0) THEN unutilized abstraction(c)

FIGURE 3. Individual representation.

Genetic programming is a heuristic search-based approach that explores the search space to find a best-fitted solution for a specific problem [25].

The abstract view of the applied genetic algorithm is shown in Figure 2. The algorithm takes quality metrics and bad smell examples as inputs and yields the best solution that corresponds to a set of detection rules. Lines 1-2 form the initial population of a genetic algorithm. An individual is represented by a set of rules with corresponding bad smells. The summation of all individuals formulates a population. Lines 4-13 represents the genetic algorithm loop. It explores the search space and constructs the new population. The fitness values of individuals are evaluated in each iteration. The best-fitted solution is saved as the best solution in each iteration (line 9). The new population is generated by selecting the best-fitted individuals from the existing population which are subsequently exposed to crossover and mutation operations. During crossover, the selected pair of parents produces two new individuals. The diversity in the solution is achieved through the mutation operation. At the end, the algorithm returns the best solution containing rules that can identify the maximum defects in a model.

A set of rules comprises an individual having IF-THEN conditional statements. The expressions in the rules are a combination of OR and AND logical operators. If a conditional statement is true, the corresponding bad smell is detected by the rule. An individual is composed of three rules with each rule looking for a specific bad smell. An instance of an individual representation is shown in Figure 3. If the number of associations (NAss), the number of invocations (NInvoc), the number of received messages (NRec) and the number of coupled classes (CBO) of a class in a sequence diagram equals or exceeds the specified thresholds, then the specified security bad smell exists in the given sequence diagram. Only the description of the individual formulation is presented here, and the definitions of the quality metrics are presented in section 4.

The number of individuals is dependent on the number of rules, which further depends on the existence of bad smells. This means that the greater the number of rules, the more individuals can be devised. The initial population is formed by the union of all the individuals. Therefore, the size of the population is indirectly contingent upon the quantity of the rules.

For crossover and mutation, the selection is based on the relative fitness of the individuals. In each iteration, the fitness value is calculated for every individual and two-third of the relatively best-fitted individuals are selected. The remaining one-third is discarded in each iteration. The discarded one-third of the population is regenerated from the selected two-thirds of the population through crossover and mutation.

For crossover, one of the three rules from an individual is randomly selected and swapped with a rule (measuring the same bad smell) in another individual, resulting in two new individuals. For example, if two individuals I1 and I2 are randomly selected for crossover, R1 in I1 will be swapped with R1 of I2. This swapping leads to the introduction of two new individuals I1‘ and I2‘. I1‘ has R1 of I2, and R2 and R3 of I1, whereas, I2‘ has R2 and R3 of I2 and R1 of I1. This creates new children (I1‘ and I2‘) which possess information from both parents (I1 and I2).

Mutation is achieved by modifying the value of the quality metrics. The algorithm randomly selects an individual, followed by a rule and then a metric, whose value will be changed. The modification to the metric value is in the form of a random increase by one or a decrease by one.

The quality of an individual is only indicated by how well the encapsulated rules have performed in detecting security bad smells. In this regard, the fitness function calculates the number of detected bad smells against the existing bad smells in a model. If a rule is able to detect a bad smell, a value of one is added to its individual's fitness; otherwise, zero is added to its fitness value. If an individual is able to detect all the defects present in a sequence diagram, the fitness value of that individual is maximized. The individuals with relatively greater fitness values are selected for crossover and mutation operations.

C. CORRECTION APPROACH

The correction of security bad smells is achieved through model transformation using XMI [26]. XML provides sufficient information in the form of tags about the transformed model. The quality metrics are extracted from the XML representations of the investigated sequence diagrams. The XML representations of sequence diagrams are modified according to the refactoring strategies against the identified bad smells. The detected security bad smells in the sequence diagrams can be traced in the corresponding XML representation. The tags are then modified to remove the detected smells. Once the refactoring is successfully applied, the refactored XML representations are exported back to the corresponding sequence diagrams. The resulting sequence diagrams will no longer have security bad smells.

D. BEHAVIORAL CONSISTENCY

Consistency can be checked via pre-conditions or post-conditions or both. The consistency approach adopted in this study is based on post-conditions. We formulate some conditions before refactoring and once the refactoring is performed, the conditions are validated. For instance, when the refactoring strategy deletes a class to correct an unutilized abstraction bad smell, it is checked to ensure that its behavior remains consistent. Since unutilized abstraction already affects behavior, the removal of the class contributes to the correctness of behavioral execution. An illustration of how corrections to security bad smells are validated in terms of behavioral consistency is presented in section 5 for each investigated sequence diagram.

E. SECURITY IMPROVEMENT VALIDATION

The assessment of security improvement in sequence diagrams due to refactoring is achieved through the statistical analysis of quality metrics. The specified quality metrics for each sequence diagram are calculated before and after refactoring, allowing the change in the metrics values to be observed. How significant the change in quality metrics is due to refactoring can only be assessed through statistical analysis. For this purpose, the pair-wise t-test is performed. The pair-wise t-test is chosen because it reflects the significant change in a pair of values. Since we have metric values before and after refactoring, it is an appropriate statistical test to execute, which can suggest significant improvement in security because of refactoring. The security improvement validation is performed in section 5.

IV. THE EXPERIMENT

The purpose of this section is to provide details on the application of the detection and correction approaches on the considered sequence diagrams to achieve the goals of this research. This section also explains the experiment setup and presents the results. The explanations related to the experiments are presented according to the guidelines provided by Jeditschka *et al.* [27].

A. EXPERIMENT GOALS

The main experimental goal is presented below in the form of the GQM (Goal Question Metric) approach [28]. The goal is to:

"Analyze the model refactoring security bad smells for the purpose of improving the quality of sequence diagrams with respect to security".

The achievement of the main goal can be broken down into multiple sub-goals. The sub-goals include successful detection and correction of security bad smells and to what extent refactoring can improve software in terms of security. To reiterate, the following are the research questions and the experiments address each:

RQ1: To what extent can the proposed detection approach detect security bad smells in sequence diagrams?

RQ2: To what extent can the proposed correction approach rectify security bad smells in sequence diagrams?

RQ3: To what extent can refactoring security bad smells improve the security aspects of sequence diagrams?

The basic mechanisms to answer these research questions are as follows:

For RQ1, examples of existing security bad smells along with quality metrics are used to evaluate the recall of the proposed detection approach.

For RQ2, the correction efficacy is computed in terms of how many security bad smells are eradicated by the correction approach.

For RQ3, we use the t-test statistical analysis of quality metrics.

B. EXPERIMENT MATERIALS

Five sequence diagrams belonging to five different systems are used for the detection purpose. The diagrams were gathered from an online source [1]. The investigated security bad smells in the sequence diagrams are: missing modularization, broken modularization and unutilized abstraction. Multiple instances (total of 20) of these three security bad smells can be seen in the investigated sequence diagrams. Multiple instances of the same bad smell allow diversity in the generated rules, which contributes to solution effectiveness.

Figure 4 shows the sequence diagram of an airline reservation system [1]. The diagram comprises five classes with associations among them except the ''Reservation System'' class. This class is assumed to be an unutilized abstract class. The presented airline reservation system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing *modularization:*

SB1: ''Customer'' class has a lot of associations and in-out calls or messages.

Broken *modularization:*

SB2: ''Flight'' class has just one received call.

Unutilized *abstraction:*

SB3: ''Reservation System'' is unassociated with any other class.

Figure 5 shows the sequence diagram of a hotel management system [1]. The diagram comprises nine classes with associations among them except the ''Staff'' class. This class is assumed to be an unutilized abstract class. The presented hotel management system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing *modularization:*

SB4: ''Receptionist'' class has a lot of associations and in-out calls or messages.

SB5: ''Customer'' class has a lot of associations and in-out calls or messages.

Broken *modularization:*

SB6: ''Stock'' class has just one received call.

SB7: ''Food Items'' class has just one received call.

SB8: ''Room Attendant'' class has just one received call.

Unutilized *abstraction:*

SB9: ''Staff'' is unassociated with any other class.

Figure 6 shows the sequence diagram of a library management system [1]. The diagram comprises six classes with associations among them except the ''Staff'' class. This class is assumed to be an unutilized abstract class. The presented library management system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing modularization:

SB10: ''Librarian'' class has a lot of associations and in-out calls or messages.

SB11: ''User'' class has a lot of associations and in-out calls or messages.

Broken *modularization:*

SB12: ''Manager'' class has just one received call.

Unutilized *abstraction:*

SB13: ''Staff'' class is unassociated with any other class.

Figure 7 shows the sequence diagram of an online movie ticketing system [1]. The diagram comprises eight classes with associations among them except the ''Visitor'' and ''Ticket'' classes. These classes are assumed to be unutilized abstract classes. The presented online movie ticketing system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing *modularization:*

SB14: ''Registered User'' class has a lot of associations and in-out calls or messages.

Broken *modularization:*

SB15: ''Cancel Ticket'' class has just one received call.

Unutilized *abstraction:*

SB16: ''Visitor'' class is unassociated with any other class.

SB17: ''Ticket'' class is unassociated with any other class.

Figure 8 shows the sequence diagram of a school management system [1]. The diagram comprises five classes with

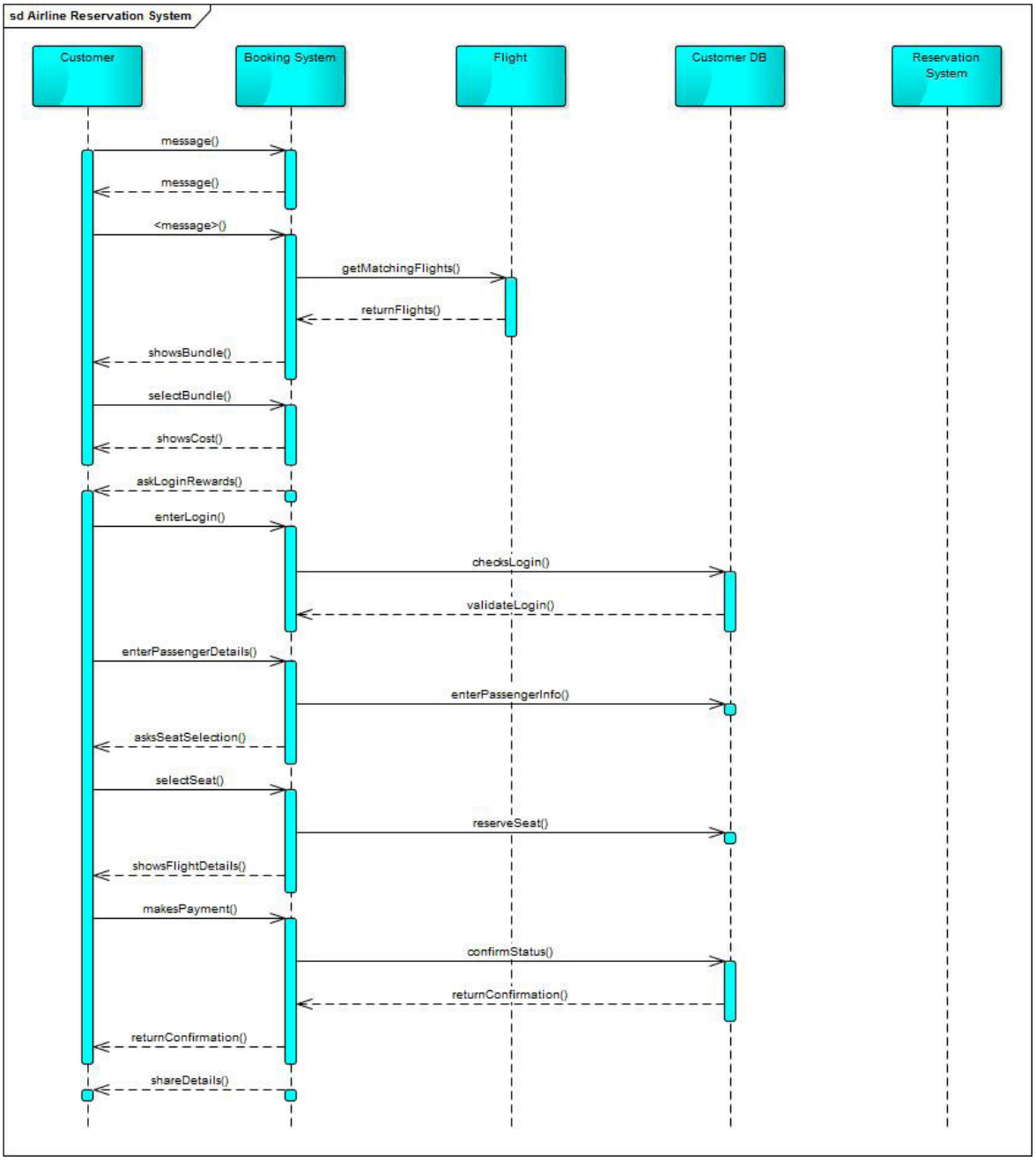

FIGURE 4. Sequence diagram of airline reservation system [1].

associations among them except the ''Employee'' class. This class is assumed to be an unutilized abstract class. The presented school management system carries three security bad smells: missing modularization, broken modularization and unutilized abstraction. The instances where these bad smells occur are listed as follows:

Missing *modularization:*

SB18: ''Admin'' class has a lot of associations and in-out calls or messages.

Broken *modularization:*

SB19: ''Class'' class has just one received call.

Unutilized *abstraction:*

SB20: ''Employee'' class is unassociated with any other class.

C. VARIABLES

The independent variable for the detection of security bad smells in the investigated sequence diagrams is detection recall. For correction, the independent variable is computed as how many security bad smells are removed as a result of

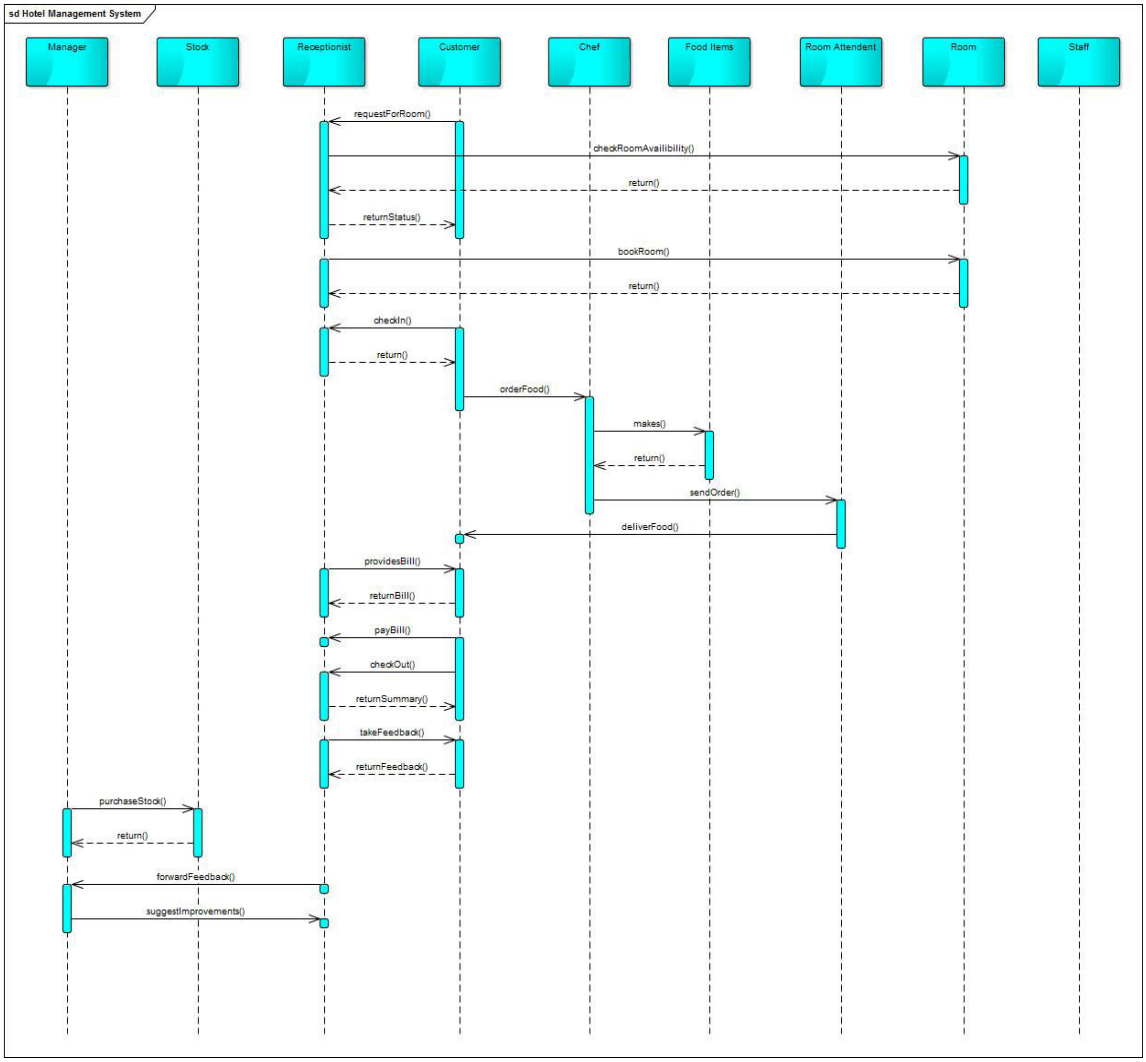

FIGURE 5. Sequence diagram of hotel management system [1].

refactoring. The other independent variable is quality metrics. The metrics are useful in the quantitative validation of security improvement. The quality metrics selected for the sequence diagrams are as follows:

- NAss is the number of in-out messages or calls for a class.
- NInvoc is the number of invoked calls for a class.
- NRec is the number of received messages for a class.
- CBO is the number of coupled classes for a class.

D. PROPOSED HYPOTHESES

The following hypotheses are formulated to statistically validate the effectiveness of the proposed approaches in order to enable a statistical judgment to be made on security improvement in sequence diagrams:

Hypothesis 1 (RQ1): The proposed detection technique is able to identify a significant number of security bad smells in the investigated sequence diagrams.

Null Hypothesis (H_{01}): The detection approach is unable to identify a significant number of security bad smells in the investigated sequence diagrams as indicated by its recall.

Alternate Hypothesis (H_{11}): The detection approach is able to identify a significant number of security bad smells in the investigated sequence diagrams as indicated by its recall.

The null hypothesis (H_{01}) is rejected in the case, where, the Detection Recall (DR) of the detection technique in terms of identifying the security bad smells in the investigated sequence diagrams is significant. The quantification of the formulated hypothesis is necessary for later testing. The quantification of the hypothesis is presented as follows in terms of detection recall:

Null Hypothesis (H_{01}): DR $< 75\%$

Alternate Hypothesis (H_{11}): DR $>= 75\%$

Hypothesis 2 (RQ2): The proposed correction technique is able to remove a significant number of security bad smells in the investigated sequence diagrams.

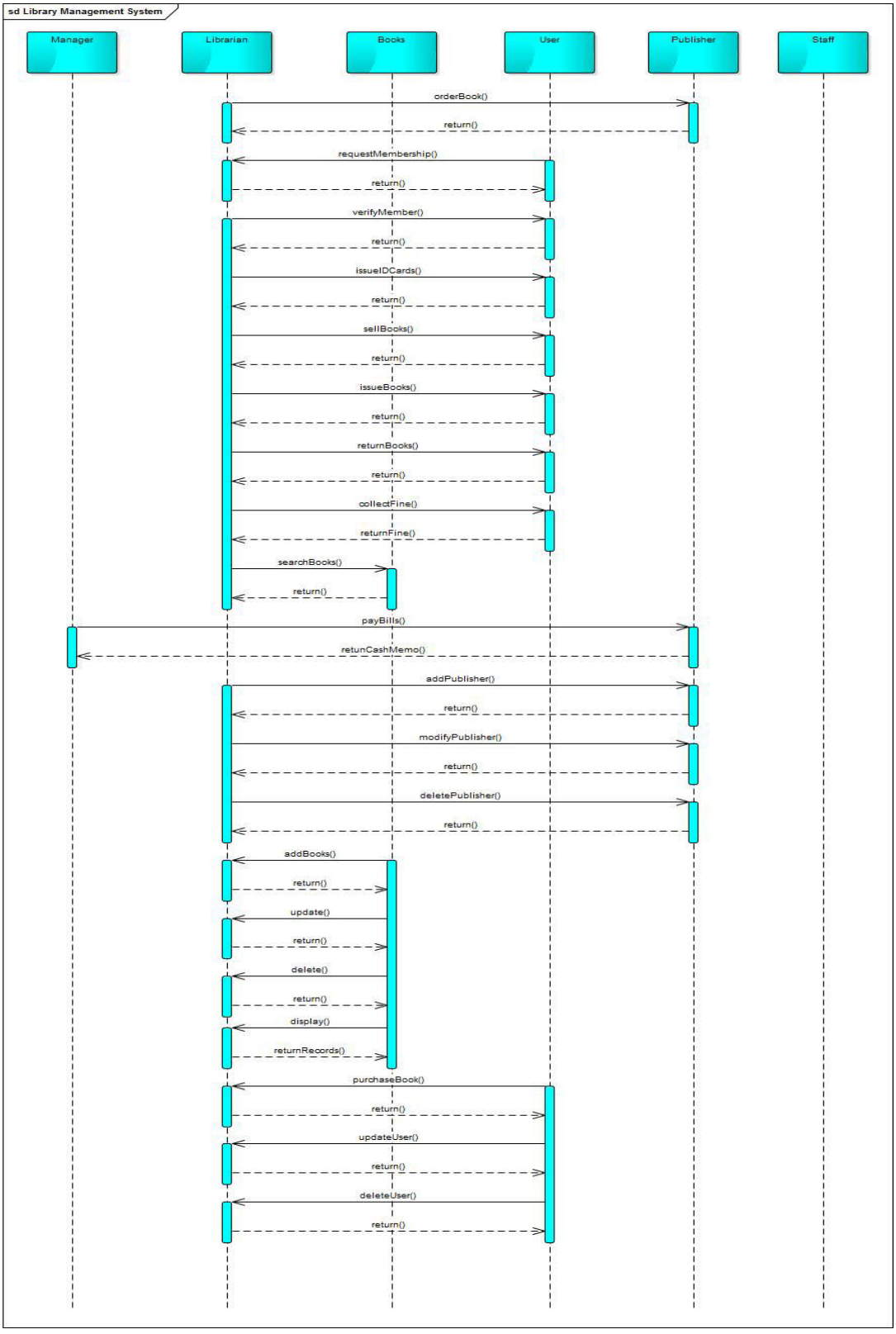

FIGURE 6. Sequence diagram of library management system [1].

Null Hypothesis (H_{02}): The correction approach is unable to remove a significant number of security bad smells in the investigated sequence diagrams as indicated by its correction effectiveness.

Alternate Hypothesis (H_{12}): The correction approach is able to remove a significant number of security bad smells in the investigated sequence diagrams as indicated by its correction effectiveness.

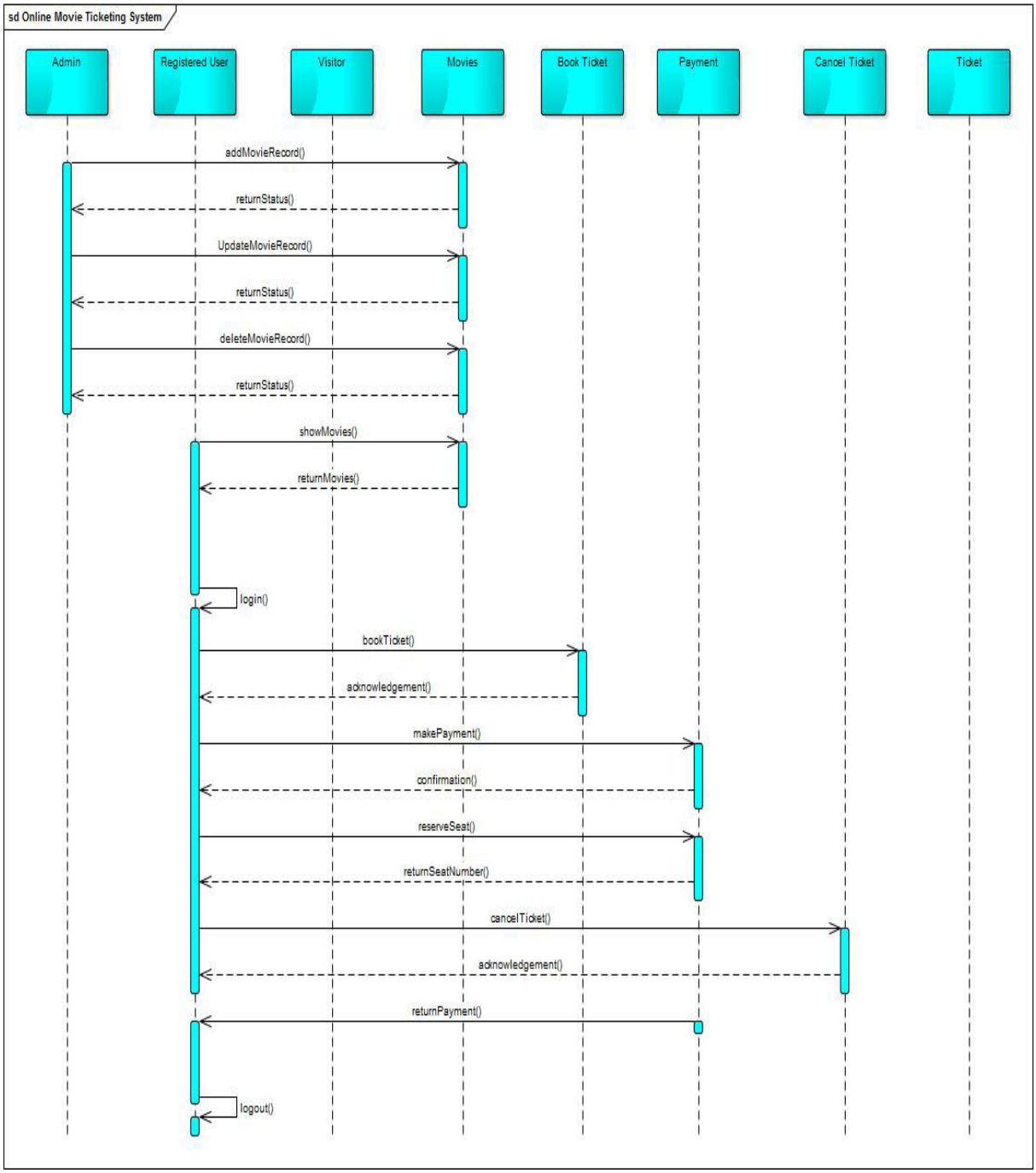

FIGURE 7. Sequence diagram of online movie ticketing system [1].

The null hypothesis (H_{02}) is rejected in the case, where, the Correction Efficacy (CE) of the correction technique in terms of removing the security bad smells in the investigated sequence diagrams is significant. The quantification of the formulated hypothesis is necessary for later testing. The quantification of the hypothesis is presented as follows in terms of correction efficacy:

Null Hypothesis (H_{02}): CE $< 75\%$

Alternate Hypothesis (H_{12}): CE $>= 75\%$

Hypothesis 3 (RQ3): Refactoring security bad smells improves the investigated sequence diagrams from a security perspective.

Null Hypothesis (H_{03}): No difference is observed in the security quality of the investigated sequence diagrams as a result of refactoring security bad smells as indicated by the quality metrics.

Alternate Hypothesis (H_{13}): A significant difference is observed in the security quality of the investigated sequence

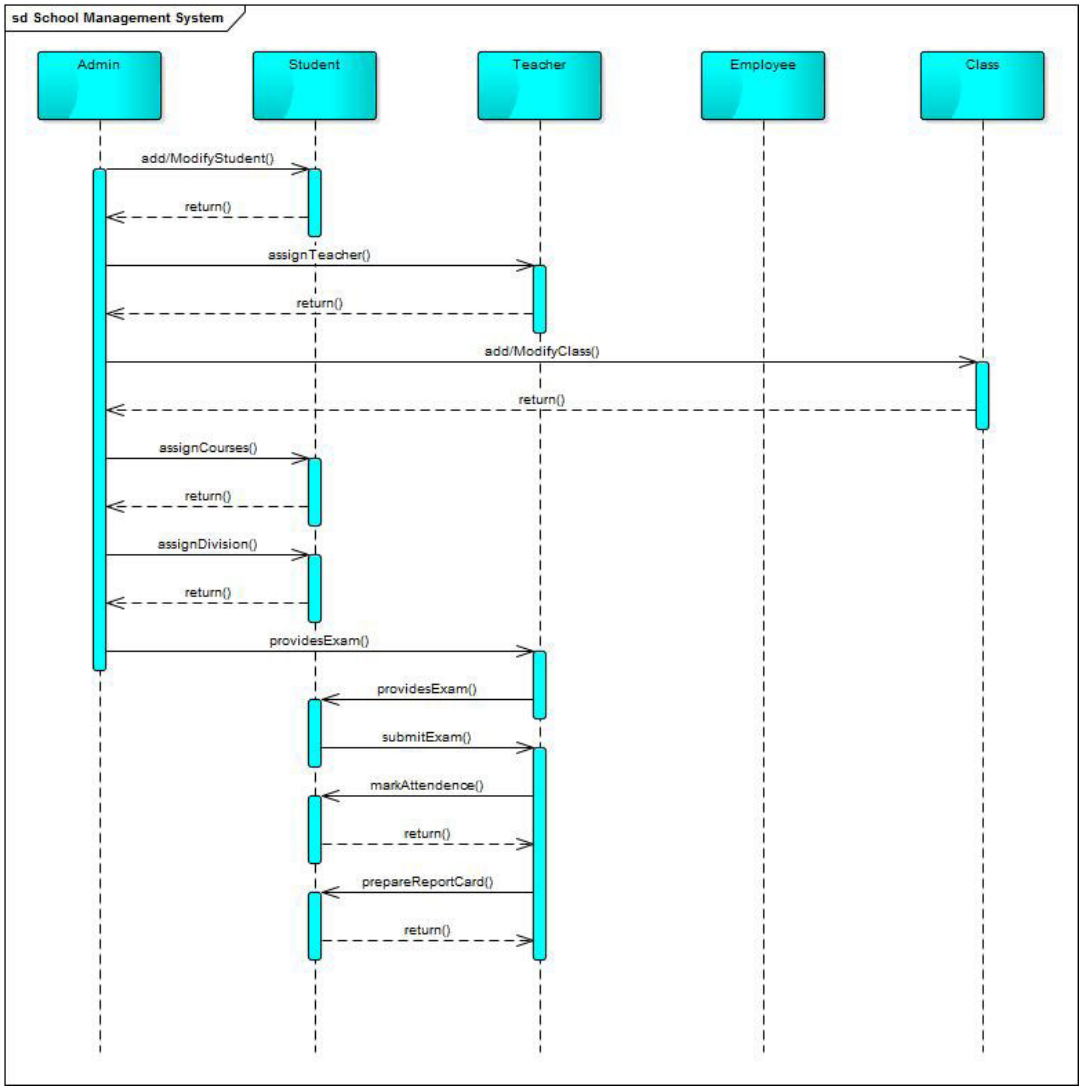

FIGURE 8. Sequence diagram of school management system [1].

diagrams as a result of refactoring security bad smells as indicated by the quality metrics.

The null hypothesis (H_{03}) is rejected in the case, where, the quality metrics values before refactoring are not equal to the quality metrics values after refactoring. The quantification of the formulated hypothesis is necessary for later testing. The quantification of the hypothesis is presented as follows in terms of p-value:

Null Hypothesis (H_{03}): p-value > 0.05

Alternate Hypothesis (H_{13}): p-value < 0.05

E. EXPERIMENT TASKS

Detection: The initial individuals are formed by governing rules from existing security bad smells in five sequence diagrams. The aggregation of individuals creates the initial population. Once the genetic algorithm reaches its terminating condition, it yields a solution which has the best fitness.

Correction: The corrections in the sequence diagrams are achieved by applying relevant refactoring techniques (listed in Appendix) to the identified security bad smells. The investigated sequence diagrams are exported using XML to perform refactoring by modifying the XML representation. For example, in the Airline Reservation System (Figure 4), there exists a security bad smell ''unutilized abstraction'', where the ''Reservation System'' abstract class is not utilized at all. This smell needs to be eradicated by removing the ''Reservation System'' abstract class from the sequence diagram. The system is first exported to XML representation and then the tags related to this abstract class are removed manually. The abridged version of the corresponding XML representation is shown in Figure 9. The tags 'lifeline' and 'element' belong to the unutilized abstract reservation system class. These tags are removed from the XML representation to remove the unutilized abstraction associated with the 'Reservation System'. The other security bad smells are removed

```
<lifeline xmi type="uml: Lifeline"
xmi:id="EAID_E54D31B3_1457_4df6_9D78_9B38F786C
C65" name="Reservation System"
visibility="public"
represents="EAID_AT000000_1457_4df6_9D78_9B38F
786CC65"/>
<element
xmiidref="EAID_E54D31B3_1457_4df6_9D78_9B38F78
6CC65" xmi: type="uml: Sequence"
name="Reservation System" scope="public">
       <model
package="EAPK_AD88F14D_A162_48d9_B68E_2EDFFFA5
1759" tpos="0" ea_localid="20"
ea_eleType="element"/>
       <properties isSpecification="false"
sType="Sequence" nType="0" scope="public"/>
       <project author="hp" version="1.0"
phase="1.0" created="2016-06-13 13:04:44"
modified="2016-06-13 13:04:53" complexity="1"
status="Proposed"/>
       <code gentype="&lt; none&gt;"/>
       <style appearance="BackColor=-1;
BorderColor=-1; BorderWidth=-1; FontColor=-1;
VSwimLanes=1; HSwimLanes=1; BorderStyle=0;"/>
       <tags/>
       <xrefs/>
       <extendedProperties tagged="0"
package_name="Sequence Model"/>
</element>
```

FIGURE 9. Abridged XML of airline reservation system.

using the same process through the related refactoring techniques specified in Appendix.

Behavioral Preservation: Behavioral consistency for sequence diagrams is verified manually. The unutilized abstraction does not contribute to the sequence diagram, so refactoring this bad smell does not introduce any behavioral inconsistency issue. Refactoring broken modularization moves the method to the class that it needs. Previously, the class called it from another class and violated multiple security attributes. After refactoring, the functionality is moved to the class which was calling it from another class, leaving the behavior untouched. Refactoring missing modularization decomposes a class into two classes and distributes the relevant functionalities according to their concerns. The most important post-refactoring condition to meet is the presence of all functionalities after decomposition. In this case, the semantics are present and require the involvement of the designer. Another condition to meet is that the interactions of the decomposed class with the other classes remain intact. In other words, the sending and receiving of messages between the refactored class and the other classes should continue unchanged.

F. RESULTS

Detection: Once the detection technique is applied on sequence diagrams, it generates the best solution. The set of rules representing the best solution yielded by the genetic

```
R1: IF (NAss(c) >= 11 AND (NInvoc(c) >= 6
AND NRec(c) >= 5) AND CBO(c) >= 3) THEN
missing modularization(c)
R2: IF (NAss(c) == 2 AND (NInvoc(c) == 1 OR
NRec(c) == 1) AND CBO(c) == 1) THEN broken
modularization(c)
R3: IF (NAss(c) == 0 AND NInvoc(c) == 0 AND
NRec(c) == 0 AND CBO(c) == 0) THEN
unutilized abstraction(c)
```

FIGURE 10. Best solution generated for sequence diagrams.

TABLE 2. Applied refactoring of airline reservation system.

Security bad smell ID	Applied refactoring
SB1	Extract Visitor class from Customer class and move relevant methods to it
SB2	Move method to Booking System and remove Flight class
SB3	Remove reservation system abstract class

TABLE 3. Applied refactoring of hotel management system.

Security bad smell ID	Applied refactoring
SB4	Extract Assistant class from Receptionist class and move relevant methods to it
SB5	Extract Resident class from Customer class and move relevant methods to it
SB6	Move method to Manager class and remove Stock class
SB7	Move method to Chef class and remove Food Items class
SB8	Move method to Chef class and remove Room Attendant class
SB9	Remove Staff abstract class

algorithm is shown in Figure 10. All three rules measure bad smells using four conditional statements with variables: NAss, NInvoc, NRec, and CBO. R1, R2 and R3 measure missing modularization, broken modularization and unutilized abstraction, respectively. The considered quality metrics and the corresponding mapping of rules to specific bad smells are extracted from Fourati *et al.* [11]. If the metrics values of class(c) equal or exceed the thresholds provided by these rules, then that class has a corresponding bad smell. The best solution (shown in Figure 10) is then applied on the investigated sequence diagrams to evaluate its recall effectiveness.

TABLE 4. Applied refactoring of library management system.

Security bad smell ID	Applied refactoring
SB10	Extract Assistant class from Librarian class and move relevant methods to it
SB11	Extract Premium User class from User class and move relevant methods to it
SB12	Move method to Publisher class and remove Manager class
SB13	Remove Staff abstract class

TABLE 5. Applied refactoring of online movie ticketing system.

Security bad smell ID	Applied refactoring
SB14	The smell is automatically removed by refactoring other smells
SB15	Move method to Registered User class and remove Cancel Ticket class
SB16	Remove Visitor abstract class
SB17	Remove Ticket abstract class

TABLE 6. Applied refactoring of school management system.

Security bad smell ID	Applied refactoring
SB18	Extract Department class from Admin class and move relevant methods to it
SB19	Move method to Admin class and remove Class *class*
SB20	Remove Employee abstract class

The set of rules governing the best solution is able to identify 15 out of 20 security bad smells present in the five examined sequence diagrams, meaning the detection approach has 75% recall. The recall is validated manually as well to confirm the correct detection of security bad smells. This result for recall provides sufficient evidence to answer RQ1 that the proposed detection approach can detect a significant number of security bad smells in sequence diagram.

Correction: The same procedure is applied to remove security bad smells in the other investigated sequence diagrams. Table 2 to Table 6 summarize the refactoring application to identify the security bad smells in the sequence diagrams of the Airline Reservation System, the Hotel Management System, the Library Management System, the Online Movie Ticketing System and the School Management System, respectively. These sequence diagrams after refactoring are shown in Figure 11 to Figure 15, respectively. Of the 20 security bad smells, 19 are eradicated through the correction approach, giving a 95% correction effectiveness.

G. HYPOTHESES TESTING

We formulated three hypotheses to address the corresponding three research questions. Each hypothesis focuses on a specific research question. The hypotheses are numerically validated as follows:

Hypothesis 1 (RQ1): In order to test this hypothesis, the Detection Recall (DR) of the detection approach is measured. The null hypothesis (H_{01}) can be rejected if DR is significant. Numerically, it is set that if DR is greater than or equal to 75%, then the null hypothesis (H_{01}) can be rejected. While being executed on the investigated sequence diagrams, the detection approach shows a significant DR of 75%. The DR is equal to 75%, so the null hypothesis (H_{01}) is rejected. This answers RQ1, that the proposed detection approach is able to detect a significant number of the security bad smells in sequence diagrams.

Hypothesis 2 (RQ2): In order to test this hypothesis, the Correction Efficacy (CE) of the correction approach is measured. The null hypothesis (H_{02}) can be rejected if CE is significant. In numerical terms, if CE is greater than or equal to 75%, then the null hypothesis (H_{02}) can be rejected. The correction approach shows notable results by yielding a significant CE of 95% in the investigated sequence diagrams. The CE is greater than 75%, so the null hypothesis (H_{02}) is rejected. This addresses RQ2, that the proposed correction approach is able to remove 95% of the security bad smells in sequence diagrams.

Hypothesis 3 (RQ3): The pair-wise t-test is performed to statistically analyze the significant security improvement in sequence diagrams. The pair-wise t-test is beneficial because it is able to identify the differences in quality metrics as a result of refactoring. The p-value is computed with 95% confidence through a pair-wise t-test. The computed p-value is 0.04, which is less than 0.05. Hence, it can be concluded that security in the investigated sequence diagrams has improved significantly. Subsequent to this observation, we can reject the formulated null hypothesis (H_{03}) with 95% confidence. This accomplishes the sub-goal of security improvement in sequence diagrams and answers RQ3. For reference, the

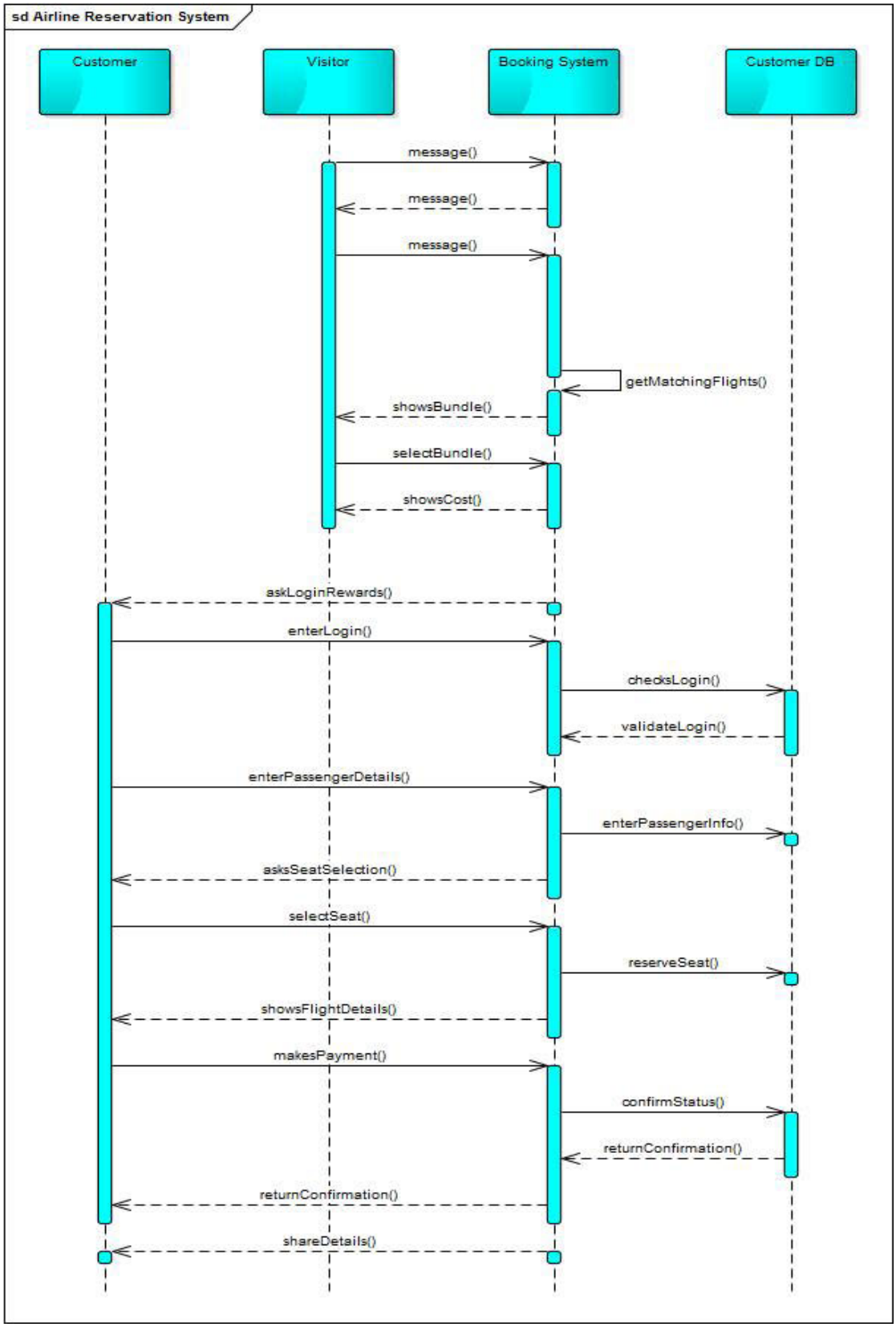

FIGURE 11. Refactored sequence diagram of airline reservation system.

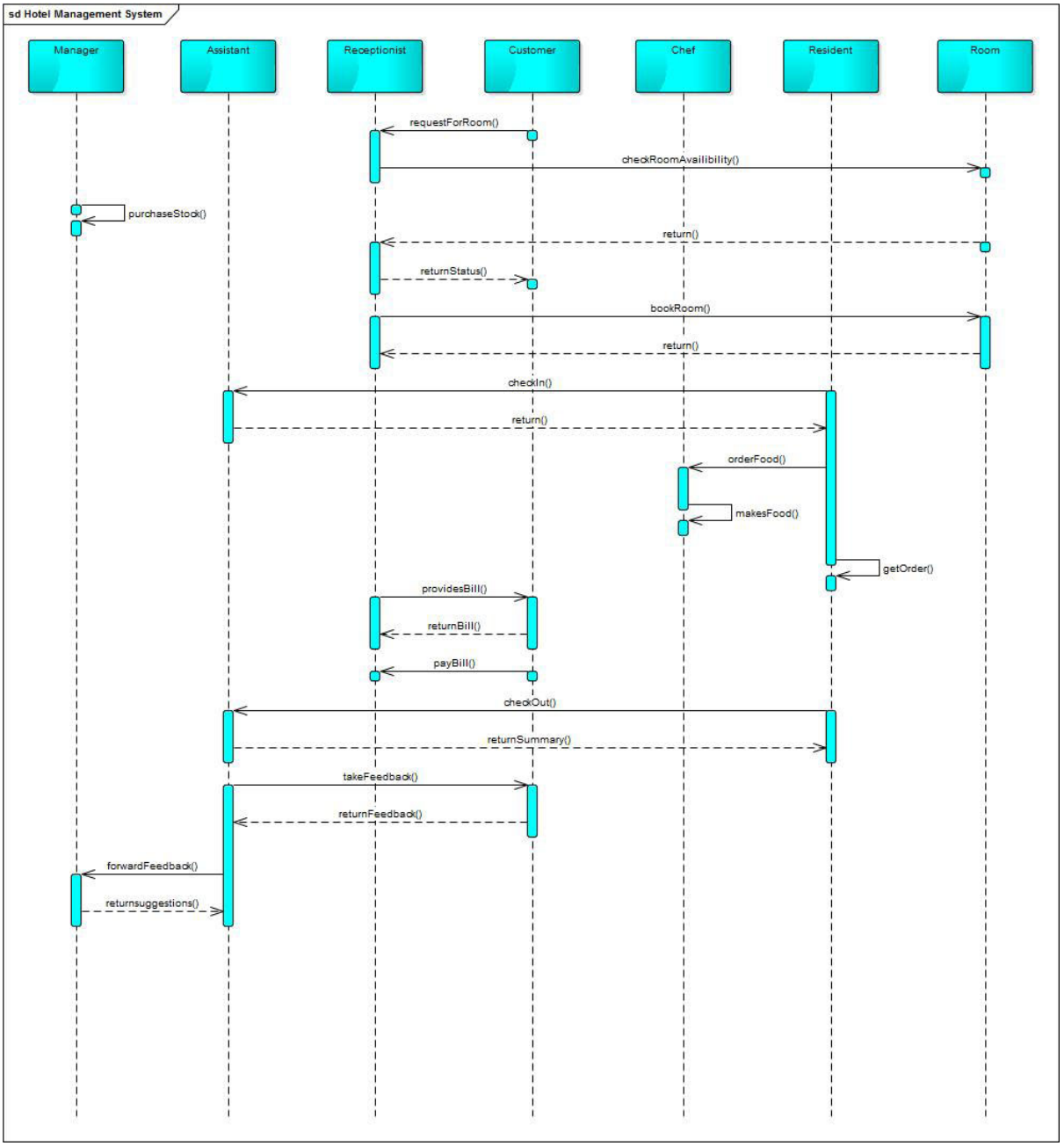

FIGURE 12. Refactored sequence diagram of hotel management system.

quality metrics values of the five investigated sequence diagrams (pre- and post-refactoring) are provided in Table 7.

H. EXPERIMENTS WITH LARGE DATASETS

The main purpose of the supplementary experiments is to gain further confidence in the detection approach in terms of generating the set of rules. To further increase confidence in the detection approach, we conduct experiments with large datasets. The abundance of security bad smells can strengthen the applicability of the generated detection rules because the generation of detection rules relies on them. The supplementary experiments address this notion and justify the generalization of detection rules. The supplementary experiments are performed on sequence diagrams which are much larger in size. This data is artificially generated because of the unavailability of data of a significant size. The generation of data is performed in two ways: 1) simple replication and 2) varied replication.

The large datasets contribute to improving the generalization of rules and the diversity of security bad smells.

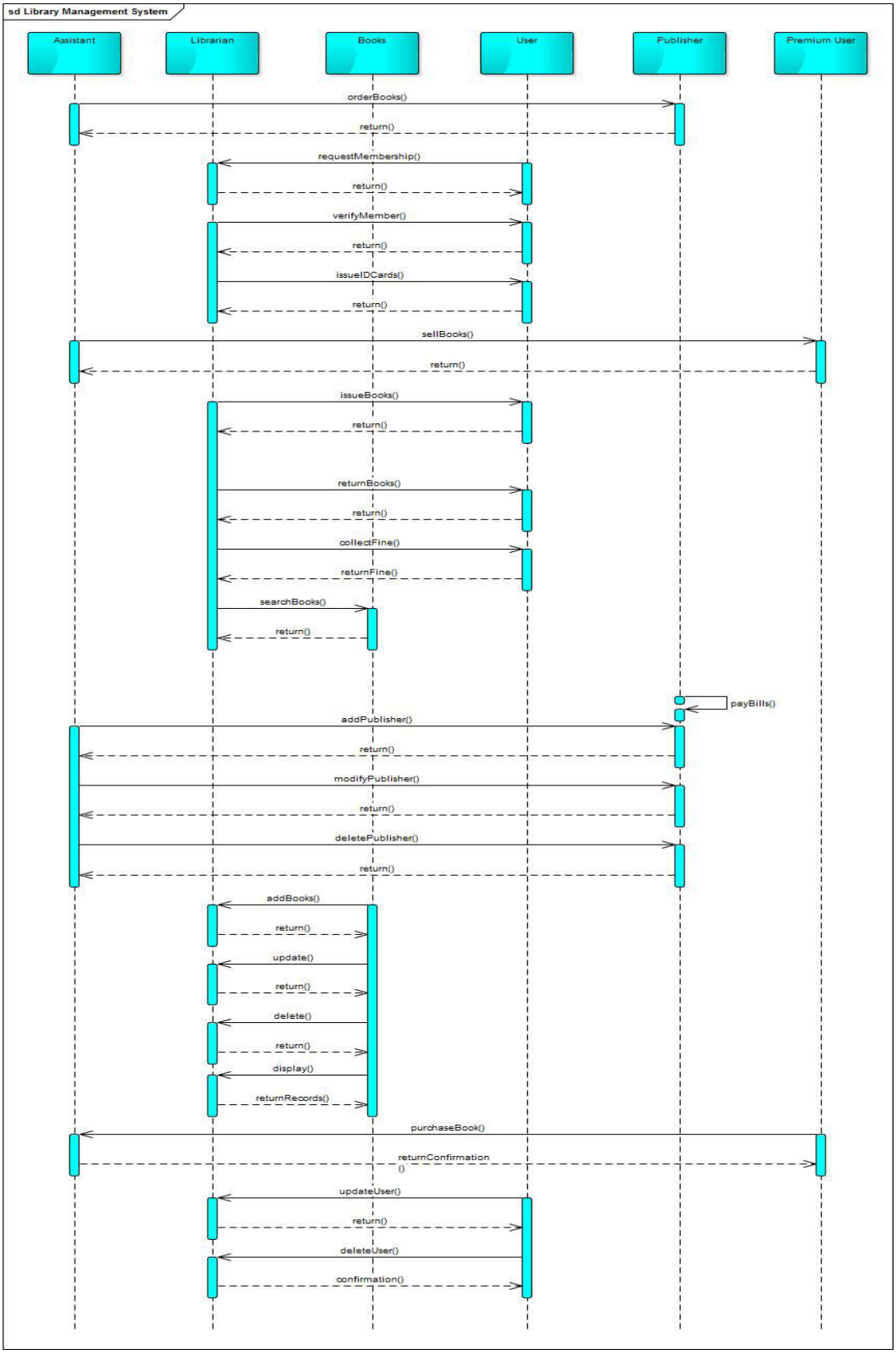

FIGURE 13. Refactored sequence diagram of library management system.

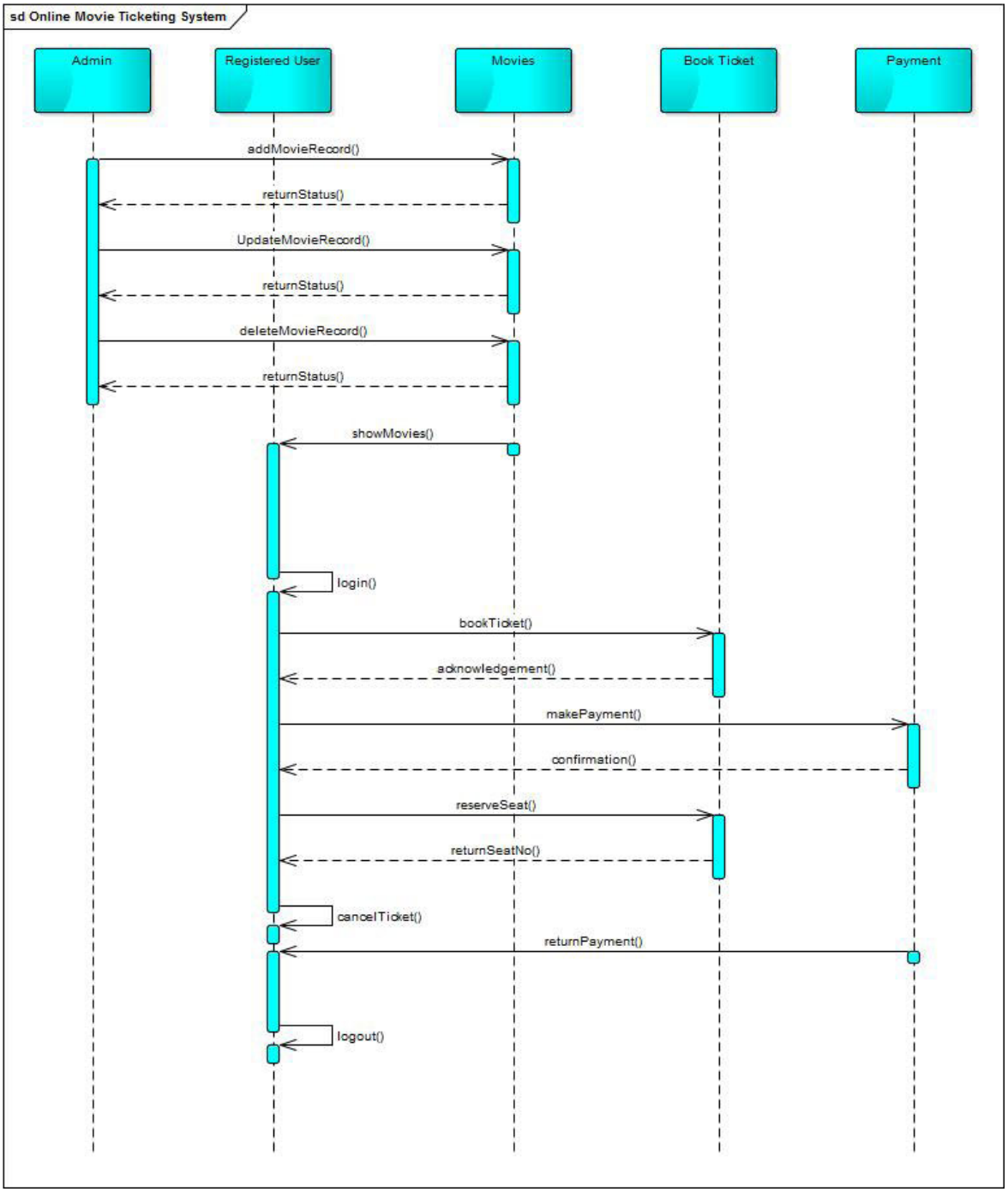

FIGURE 14. Refactored sequence diagram of online movie ticketing system.

The detection rules are generated once the large datasets are input to the GA. The variations between new and old solutions need to be analyzed. In addition, the improvement in the detection rules generated by the supplementary experiments needs to be assessed. The improvement in detection rules is assessed by either enhanced detection recall or refinement of the rules in general.

1) EXPERIMENTS WITH SIMPLE REPLICATION DATASETS

In simple replication, the data is produced by replicating the small datasets. For example, we start with existing five sequence diagrams, then create the replications of these diagrams. The replication process halts when at least 1000 sequence diagrams are created. Table 8 shows the statistics of the simple replicated datasets for the investigated

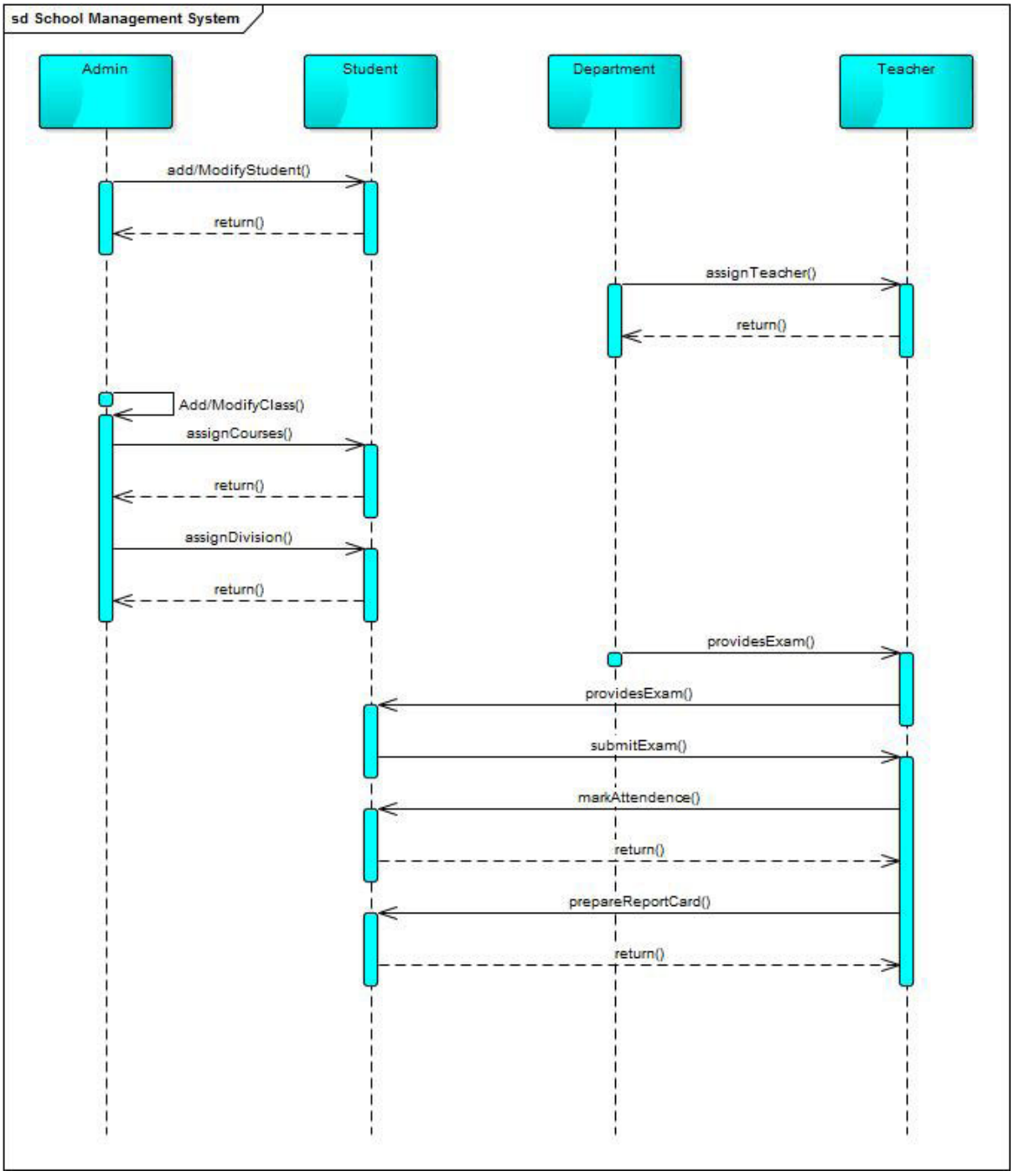

FIGURE 15. Refactored sequence diagram of school management system.

sequence diagrams. Data size is significantly enhanced in terms of the number of lifelines and security bad smell instances. Since the data size is increased, the security bad smell instances are automatically escalated.

The set of rules generated by the supplementary experiment with simple replication differs marginally from the best solution stated earlier. Figure 16 states the best-generated solution as a result of the supplementary experiment. The differences are observed in R1, while R2 and R3 remain unmodified. A decrease is observed in the NInvoc and NRec values. The new best solution incorporates a lack of coupling in the rule and as a result, the DR becomes 90%.

2) EXPERIMENTS WITH VARIED REPLICATION DATASETS

The reason for replicating with variations is to introduce more distinctive quality metrics values. For example, we start with

TABLE 7. Quality metrics values pre- and post-refactoring.

System	Class	Before refactoring				After refactoring			
		NAss	NInvo	NRec	CBO	NAss	NInvo	NRec	CBO
Airline Reservation	Customer	15	7	8	1	9	4	5	1
	Booking System	23	5	3	3	21	5	2	3
	Flight	2	0	1	1				
	Customer DB	6	0	4	1	6	0	4	1
	Reservation System	0	0	0	0				
	Visitor					6	3	3	1
Movie Ticketing	Admin	6	3	3	1	6	3	3	1
	Registered User	11	5	6	4	8	3	4	3
	Visitor	0	0	0	0				
	Movies	8	0	4	2	7	1	3	2
	Book Ticket	4	0	2	1	4	0	2	1
	Payment	3	1	1	1	3	1	1	1
	Cancel Ticket	2	0	1	1				
	Ticket	0	0	0	0				
School Management	Admin	11	6	5	3	6	3	3	1
	Student	12	1	6	2	12	1	6	2
	Teacher	9	3	5	2	9	3	5	2
	Employee	0	0	0	0				
	Class	2	0	1	1				
	Department					3	2	1	1
Library Management	Librarian	38	11	19	3	26	6	13	2
	Books	10	4	5	1	10	4	5	1
	User	20	4	10	1	16	3	8	1
	Publisher	10	1	5	2	8	0	4	1
	Staff	0	0	0	0				
	Manager	2	0	1	1				
	Assistant					12	5	6	2
	Premium User					4	1	2	1
Hotel Management	Manager	4	1	2	2	2	0	1	1
	Stock	2	0	1	1				
	Receptionist	17	5	9	3	9	3	5	2
	Customer	13	5	6	3	7	2	3	2
	Chef	4	2	2	3	1	0	1	1
	Food Items	2	0	1	1				
	Room Attendant	2	1	1	1				
	Room	4	0	2	1	4	0	2	1
	Staff	0	0	0	0				
	Assistant					6	1	3	3
	Resident					5	3	2	2

TABLE 8. Statistics of simple replicated datasets.

UML model	Number of lifelines	Security bad smell instances
Sequence diagram	1023	620

R1: IF (NAss(c) >= 11 AND (NInvoc(c) >= 4 AND NRec(c) >= 4) AND CBO(c) >= 3) THEN missing modularization(c)
R2: IF (NAss(c) == 2 AND (NInvoc(c) == 1 OR NRec(c) == 1) AND CBO(c) == 1) THEN broken modularization(c)
R3: IF (NAss(c) == 0 AND NInvoc(c) == 0 AND NRec(c) == 0 AND CBO(c) == 0) THEN unutilized abstraction(c)

FIGURE 16. Best solution for sequence diagrams (simple replication).

the existing five sequence diagrams, then create replications of these diagrams and modify the metrics values. The new dataset consists of ten different sequence diagrams because of the introduction of variations. The replication procedure halts when at least 1000 sequence diagrams are created. The statistics related to the varied replication dataset are shown in Table 9. The datasets sizes are significantly increased in terms of the number of lifelines. Since the data size is increased, the security bad smell instances are automatically escalated. Another reason for enhanced bad smell instances is the introduction of new instances due to variations.

TABLE 9. Statistics of replicated datasets with variations.

UML model	Number of lifelines	Security bad smell instances
Sequence diagram	1056	640

R1: IF (NAss(c) >= 11 AND (NInvoc(c) >= 4 AND NRec(c) >= 5) AND CBO(c) >= 2) THEN missing modularization(c)
R2: IF (NAss(c) == 2 AND (NInvoc(c) == 0 OR NRec(c) == 1) AND CBO(c) == 1) THEN broken modularization(c)
R3: IF (NAss(c) == 0 AND NInvoc(c) == 0 AND NRec(c) == 0 AND CBO(c) == 0) THEN unutilized abstraction(c)

FIGURE 17. Best solution for sequence diagrams (varied replication).

Datasets created through varied replication examine the flexibility of the GA as well.

Sequence Diagrams: In the experiment with datasets produced from varied replication, two quality metrics values are modified in comparison with the best solution presented earlier. Figure 17 states the best-generated solution as a result of the supplementary experiment with the varied dataset. The varied metrics are CBO and NInvoc of R1. The change in more quality metrics supports the variations in the generated dataset. Similar to the simple replication experiment, the best solution generated from the varied replication has a DR of 90%. Therefore, we can conclude that the large datasets created from both types of replication have further refined the generated set of rules. This improvement in results contributes to strengthening the generalization of these rules.

As a conclusion, we can express that the supplementary experiments with large sets of sequence diagrams and security bad smell instances have improved a few rules in the generated solutions. The new sets of rules are also a refined form of the previously acquired rules from the small datasets in a way that they show the potential of capturing more security bad smells. This aids the generation of a more reliable and generalized set of rules.

V. ANALYSIS AND DISCUSSION

This section focuses on the implications of this research. The analysis and discussion justify many expected propositions. The justifications for the use of security bad smell examples and their abundance in the investigated sequence diagrams are presented in this section. A discussion on the quality metrics and their values is provided as well. This section also analyzes the impact of refactoring on the security of other quality attributes.

A. SECURITY BAD SMELLS

In the detection approach, the most important component is security bad smell examples because the formulation of rules mainly depends on this. The quality of the solution is contingent on the quality of the base examples. During individual formulation, diversity is ensured by selecting rules wisely. This is evident during detection validation as the yielded solution can detect a significant number of security bad smells. In addition, an abundance of investigated security bad smells has allowed us to draw solutions with the maximum recall.

Bad smell examples are in abundance in online software repositories. Sometimes, bad smells are reported in the maintenance directory, and if not, they can easily be identified manually or using existing tools. The bad smell examples incorporate the actual programming practices in the detection process. As a result, the yielded rules are more precise and context faithful. The examples also remove the existing contradictions in the metrics threshold values as it solves the subtleness of agreeing on commonly accepted metrics values. The rules generation process is executed multiple times using bad smell examples to erase any uncertainty with respect to the quality of rules. Although it is assured that a decent frequency of security bad smells exists in the investigated sequence diagrams, the number of instances varies among different bad smells. The security bad smells found in the studied sequence diagrams are evenly distributed. In other words, the frequencies of different types of bad smells are almost equal.

It can be argued that work overhead exists when using security bad smell examples because they need to be identified before the start of the GA. The rationale for using base examples is to remove any confusion about the quality metrics thresholds. It would have been a major threat to validity if thresholds were used instead of base examples. A consensus on the quality metrics thresholds would have taken this study to unjustifiable arguments. However, the use of base examples in this study is completely justified in the study. Another reason for using base examples is to incorporate real programming mistakes that lead to security bad smells. Another consideration is the dependency of the detection approach on the size of base examples set. The experiment clearly answers this argument by showing significant results using a small set of base examples.

B. CONSISTENCY OF RESULTS

Another concern to discuss is the consistency of the results. The acquired results are consistent because the detection approach incorporates quantitative information using quality metrics. If the semantics of the investigated sequence diagrams are also considered, then consistency would have been an issue to address. The approach is irrelevant to sequence diagram semantics, so the consistency of results is not a concern.

In order to improve the consistency of results, we performed experiments using large datasets of three of the investigated sequence diagrams. Although experiments with small datasets give significant results, experiments with large datasets further improve the results in terms of consistency. Consistency can be observed from the achieved detection

recall when experimenting with large datasets. The experiments also help in the refinement of the detection rules. The same is confirmed by the detection recall of the detection approach as well. The experiments with varied replications produce comparatively better detection rules. The rationale behind this is due to the diversity in the datasets because of the modifications during replication.

We believe that the approaches are stable. The generated solution from multiple runs of the GA almost yielded solutions with none or minimal difference in fitness. The sets of rules generated by experiments with large datasets also do not show a significant difference. During correction, the XML transformations from sequence diagrams remain consistent.

C. VARIATIONS IN QUALITY METRICS

During the validation of the security improvement, changes in quality metrics values are observed. Though all quality metrics contribute towards the security improvement in a specific sequence diagram, the impact of metrics may vary depending on the security bad smell being removed. For instance, refactoring missing modularization brings significant changes to the metrics values because the class undergoes decomposition. On the other hand, refactoring broken modularization marginally changes the metrics. Another important point to discuss is the trend of the variation in the metrics. It is observed that refactoring causes the metrics values to decrease. This means that it is desirable to have metrics with lower values to have a more secure sequence diagram.

D. IMPACT OF APPLIED REFACTORING ON QUALITY ATTRIBUTES

Although we analyzed the effect of refactoring on security improvement, the sequence diagrams showed improvements in other quality properties as well. The notable enhancement in quality is observed in terms of modularity, complexity, reusability and design size.

The quality improvements in sequence diagrams are also observed in a similar manner. The issue of unutilized abstraction is resolved through the removal of the abstraction. This not only decreases the design size but also ensures correct operational behavior. The problem of broken modularization is solved by the moving method and removing the respective class. The movement method strengthens the modularization and the removal of the class contributes to the reduction in the design size. The overall number of messages is also reduced. The major reduction in design size comes from the removal of lifelines as a result of refactoring unutilized abstraction and broken modularization. Refactoring missing modularization reduced the number of messages between two classes. The burden of interactions between two classes is shared by a newly introduced class. This way the model is modularized, which means less complexity and more reusability. The separation of concerns is also validated since classes now only deal with what concerns them. The two-undetected missing modularization bad smells in sequence diagrams are due to a lack of coupling. It is normally perceived that if a component has a missing modularization issue, it exhibits high coupling. However, in these two cases, the coupling was low regardless of the missing modularization problem. The low coupling restricted the rule from detecting the bad smells.

It can be observed that as a byproduct of refactoring security bad smells, other quality attributes are improved. This quality upgrade is observed in all the investigated sequence diagrams. Modularity, complexity, design size and reusability are the quality attributes that show the quality revamp. In addition, the introduction of these quality attributes eases the analysis of sequence diagrams.

E. THREATS TO VALIDITY

This section reports the validity threats and how they were mitigated to minify their impact on the experimental validation of the proposed techniques. The most common classification to address validity threats is construct validity, conclusion validity, internal validity and external validity [29], [30] and is adopted to report the validity threats of the research.

The most important activity in the experimental process is the selection of independent variables. The correlation between dependent and independent variables needs to be closely examined. In the experimental validation, the independent variables i.e. quality metrics, are selected based on previous studies and after in-depth analysis to ensure their effectiveness in measuring the security aspects in sequence diagrams. Some main security bad smell examples might be overlooked during individual formulation. This threat is mitigated because of crossover and mutation operations. The suspicions about biasness of experimental outcomes are totally removed by laying no pre-expectations on the experiments.

The conclusions drawn from the experiments are based on sufficient subjective and objective findings. The supreme objectivity of quality metrics has encouraged us to incorporate them in the empirical validation. The objectivity of quality metrics has allowed us to reach meaningful and definite conclusions. The quality metrics are manually calculated with absolute care, but there is always a threat posed by manual computation. This threat is minified by computing the quality metrics multiple times. The replication of the datasets is also performed manually. This threat is minimized by making the replication random. Randomization introduces diversity in the datasets, which is the ultimate objective.

The analyzed sequence diagrams are not exposed to any treatment except correction to observe only the influence of refactoring on them. No modifications in treatments are made to observe findings under similar conditions. The post-refactoring states of sequence diagrams are carefully saved for the computation of quality metrics. The import and export of sequence diagrams to and from XML are performed using the same tool to avoid any structural change in XML representations. The modifications in XML representations are performed manually but these do not impact validity because

TABLE 10. Taxonomy of Security Bad Smells [3], [9], [31], [32].

Bad Smell	Description	Security Violation	Refactoring
Missing Abstraction	In the absence of an abstraction, the data and behavior are spread across the code	Confidentiality, Secrecy, Guarded Access, Integrity, Insecure Info Flow.	Replace type-code with class.
Incomplete Abstraction	When an abstraction does not support complementary or interrelated methods completely.	Correctness, Integrity	Introduce the missing complementary operation(s) in the class
Multifaceted Abstraction	When abstraction is assigned more than one responsibility.	Correctness, Integrity	Extract class
Unutilized Abstraction	When the code in unused abstractions is accidentally invoked, it may result in runtime problems, affecting reliability	Integrity.	Remove unutilized abstraction.
Duplicate Abstraction	When two abstractions have the same names, it is confusing as to which abstraction to invoke.	Non-repudiation, Integrity.	In the case of name, rename one of the abstractions. In the case of implementation, if it is exactly same, remove one implementation. If slightly different, then move to a single class.
Deficient Encapsulation	It provides direct access of a class's data to outside classes.	Confidentiality, Secrecy, Guarded Access, Integrity.	Encapsulate field
Leaky Encapsulation	When internal data structures are leaked, the integrity of abstraction may be compromised.	Confidentiality, Secrecy, Guarded Access, Integrity.	Encapsulate field and methods (if necessary)
Missing Encapsulation	When implementation variations are not encapsulated.	Confidentiality, Secrecy, Guarded Access, Integrity.	Encapsulate field and methods (if necessary)
Broken Modularization	The data and related procedures are split across abstractions.	Confidentiality, Secrecy, Guarded Access, Integrity.	Move method/Field
Cyclically dependent Modularization	Changes to a cyclically dependent abstraction can lead to runtime problems across other abstractions.	Integrity	Move method or field to move the code that introduces a cyclic dependency to one of the participating abstractions.
Missing Modularization	When a class is not decomposed.	Reliability, Correctness	Extract class, Move methods
Rebellious Hierarchy	When a subtype rejects the methods provided by its super-type.	Reliability, Correctness, Integrity.	Apply move method from the super-type to the relevant subtypes
Unnecessary Hierarchy	When inheritance is applied unnecessarily for a particular design context.	Integrity, Insecure Info Flow.	Collapse hierarchy
Missing Hierarchy	To explicitly manage variation in hierarchical behavior, where a hierarchy could have been created and used to encapsulate those variations.	Reliability, Integrity	Connection with appropriate hierarchy interface should be made
Broken hierarchy	When developers are not aware that the super-type and subtype do not share an IS-A relationship.	Confidentiality, Secrecy, Guarded Access, Insecure Info Flow, Integrity.	Replace inheritance with delegation

the corresponding exported sequence diagrams are validated with the expected refactored sequence diagrams.

External validity usually poses threats to the generalization of results. To mitigate this threat, a favorable number of case studies are collected and are a good representation of actual sequence diagrams. The generalization of results is also improved because validation is also performed with large datasets carrying a significant number of security bad smell instances.

VI. CONCLUSION AND FUTURE WORK

The quality of sequence diagrams significantly affects the quality of other software artifacts. Bad smells in a sequence diagram are likely to propagate to other phases of software development. The removal of bad smells at later stages of software development is a non-trivial task and may lead to maintainability issues. It is imperative to detect and correct bad smells from sequence diagrams to avoid maintainability issues. Although there are many quality attributes reported in the literature, the focus of this paper is on security quality of sequence diagrams. There is a lack of work investigating the contribution of refactoring in improving the security aspects of sequence diagrams.

In this research, we overcome the problem of security in sequence diagrams by the application of model refactoring. The detection of security bad smells is achieved through

the adaptation of a genetic algorithm, while correction is accomplished by the model transformation approach. The detection approach uses quality metrics to formulate rules. The best set of rules generated by GA is used for the detection of security bad smells in the studied sequence diagrams. The correction approach applies a model transformation to XML representation for refactoring identified security bad smells in the investigated sequence diagrams.

The proposed approaches are validated by performing experiments with multiple case studies of sequence diagrams. The detection approach is able to detect security bad smells with 75% recall in the investigated sequence diagrams. The correction approach also shows a significant result by removing 95% of the security bad smells by applying refactoring in the investigated sequence diagrams. We also performed supplementary experiments to generate more generalized detection rules because the detection approach relies heavily on generated rules. The sets of rules generated by the supplementary experiments are improved in terms of their ability to detect more legitimate security bad smells. Through statistical analysis of the quality metrics, we are also able to conclude that there is a significant improvement in the security quality of the investigated sequence diagrams as a result of refactoring.

The compelling results delivered by the detection and correction approaches have encouraged us to extend this work in the future. We plan to apply the approaches to other sequence diagrams to gain further confidence in their applicability to other models. We also plan to apply the same approaches with a different set of bad smells and we also plan to apply the same approach to other UML diagrams.

APPENDIX

See Table 10.

REFERENCES

[1] Cinergix. (Jun. 10, 2016). *Creately*. [Online]. Available: http://creately.com/

[2] G. Booch, J. Rumbaugh, and I. Jacobson, "The unified modeling language," *Unix Rev.*, vol. 14, no. 13, p. 5, 1996.

[3] M. Fowler, K. Beck, J. Brant, and W. Opdyke, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.

[4] I. Gorton, *Essential Software Architecture*. Berlin, Germany: Springer-Verlag, 2006.

[5] *Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation*, Standard ISO/IEC 25010:2011, 2011.

[6] M. Misbhauddin and M. Alshayeb, "UML model refactoring: A systematic literature review," *Empirical Softw. Eng.*, vol. 20, no. 1, pp. 206–251, Feb. 2015.

[7] Q. H. Do, R. Bubel, and R. Hähnle, "Automatic detection and demonstrator generation for information flow leaks in object-oriented programs," *Comput. Secur.*, vol. 67, pp. 335–349, Jun. 2017.

[8] H. Mumtaz, "Software security improvement through the application of UML model refactoring," M.S. thesis, King Fahd Univ. Petroleum Minerals, Dhahran, Saudi Arabia, 2016.

[9] G. Suryanarayana, G. Samarthyam, and T. Sharma, "Design smells," in *Refactoring for Software Design Smells*, G. Suryanarayana and G. S. Sharma, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, ch. 2, pp. 9–19.

[10] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[11] R. Fourati, N. Bouassida, and H. B. Abdallah, "A metric-based approach for anti-pattern detection in UML designs," in *Computing and Information Science*. Berlin, Germany: Springer-Verlag, 2011, pp. 17–33.

[12] M. Mohamed, M. Romdhani, and K. Ghedira, "M-REFACTOR: A new approach and tool for model refactoring," *ARPN J. Syst. Softw.*, vol. 1, no. 4, pp. 117–122, 2011.

[13] H. Mumtaz, M. Alshayeb, S. Mahmood, and M. Niazi, "An empirical study to improve software security through the application of code refactoring," *Inf. Softw. Technol.*, vol. 96, pp. 112–125, Apr. 2018.

[14] S. Ruland, G. Kulcsár, E. Leblebici, S. Peldszus, and M. Lochau, *Controlling the Attack Surface of Object-Oriented Refactorings*. Cham, Switzerland: Springer, 2018, pp. 38–55.

[15] E. Song, R. B. France, D.-K. Kim, and S. Ghosh, "Using roles for pattern-based model refactoring," in *Proc. Workshop Critical Syst. Develop. UML (CSDUML)*, 2002, pp. 1–8.

[16] D.-K. Kim, "Software quality improvement via pattern-based model refactoring," in *Proc. 11th IEEE High Assurance Syst. Eng. Symp.*, Dec. 2008, pp. 293–302.

[17] Ł. Dobrzański and L. Kuźniarz, "An approach to refactoring of executable UML models," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2006, pp. 1273–1279.

[18] A. C. Jensen and B. H. C. Cheng, "On the use of genetic programming for automated refactoring and the introduction of design patterns," in *Proc. 12th Annu. Conf. Genetic Evol. Comput. (GECCO)*, Portland, OR, USA, 2010, pp. 1341–1348.

[19] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *J. Softw., Evol. Process.*, vol. 27, no. 11, pp. 867–895, Nov. 2015.

[20] D. E. Khelladi, R. Hebig, R. Bendraou, J. Robin, and M.-P. Gervais, "Detecting complex changes and refactorings during (Meta)model evolution," *Inf. Syst.*, vol. 62, pp. 220–241, Dec. 2016.

[21] J. Jürjens, *Secure Systems Development With UML*. Berlin, Germany: Springer-Verlag, 2005.

[22] *Glossary of Key Information Security Terms*. Gaithersburg, MD, USA: National Institute of Standards and Technology, 2006.

[23] M. Whitman and H. Mattord, *Principles of Information Security*. Boston, MA, USA: Cengage, 2011.

[24] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Automated Softw. Eng.*, vol. 20, no. 1, pp. 47–79, Mar. 2013.

[25] D. E. Golberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA, USA: Addison-Wesley, 1989, p. 102.

[26] *XML Metadata Interchange*. XMI, Object Manage. Group, Needham, MA, USA, 2007.

[27] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London, U.K.: Springer, 2008, pp. 201–228.

[28] V. R. B.-G. Caldiera and H. D. Rombach, "Goal question metric paradigm," *Encyclopedia Softw. Eng.*, vol. 1, pp. 528–532, 1994.

[29] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Boston, MA, USA: Ravenio, 2015.

[30] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-Experimentation: Design & Analysis Issues for Field Settings*. Boston, MA, USA: Houghton Mifflin, 1979.

[31] (Feb. 24, 2016). *Martin Fowler*. [Online]. Available: http://refactoring.com/

[32] W. H. Brown, R. C. Malveau, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. London, U.K.: Wiley, 1998.