



A requirements inspection method based on scenarios generated by model mutation and the experimental validation

Boyuan Li¹ · Xiaoxu Diao¹ · Wei Gao¹ · Carol Smidts¹

Accepted: 9 June 2021 / Published online: 3 August 2021

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract

The requirements phase is the most critical phase of the software development life cycle. The quality of the requirements specification affects the overall quality of the subsequent phases and hence, the software product. An effective and efficient method to qualify the software requirements specification (SRS) is necessary to ensure the reliability and safety of software. In this paper, a requirements inspection method based on scenarios generated by model mutation (RIMSM) is proposed to detect defects in the functional requirements of a safety-critical system. The RIMSM method models software requirements using a High Level Extended Finite State Machine (HLEFSM). A method that executes the HLEFSM model is defined. The method uncovers the behaviors and generates the outputs of the system for a given scenario. To identify an adequate set of scenarios in which the model shall be executed, an analogue to mutation testing is defined which applies to the requirements phase. Twenty-one mutation operators are designed based on a taxonomy of defects defined for the requirements phase. Mutants of the HLEFSM model are generated using these operators. Further, an algorithm is developed to identify scenarios that can kill the mutants. The set of scenarios is considered to be adequate for detecting defects in the model when all mutants generated are killed. The HLEFSM model is then executed for the scenarios generated. The results of execution are used to detect defects in the model. A Requirements Inspection Tool based on Scenarios Generated by Model Mutation (RITSM) is developed to automate the application of the RIMSM method. The performance and usability of the RIMSM method are studied and demonstrated in an experiment by comparing the RIMSM method to the checklist-based reading method.

Keywords Requirements inspection · Defects detection · Mutation testing · Scenario generation

Communicated by: Jeff Offutt

✉ Boyuan Li
li.4935@osu.edu

Extended author information available on the last page of the article

1 Introduction

The software development life cycle begins with the requirement phase. A software requirements specification (SRS) is usually delivered at the end of the requirements phase. The quality of the requirements specification affects the overall quality of the subsequent phases and hence, the software product (Alshazly et al. 2014). Internal NASA data shows that about 40% of defects identified in software products originate from requirements defects (Arndt et al. 2017). To ensure the reliability and safety of a software, reducing the number of defects in its SRS is significant.

During software development, the SRS is inspected by the Verification & Validation team to detect defects. The commonly used inspection techniques include Ad Hoc Reading, Checklist-Based Reading (CBR), Perspective Based Reading, etc. However, previous research shows that these techniques exhibit low performance. According to an experiment conducted by A. Porter et al. (2002), 36.5% of defects can be detected using Checklist-Based Reading and 32.5% of the defects can be detected using Ad Hoc Reading. In the experiment of He and Carver (2006), the Perspective-Based Reading technique exhibited a detection rate of about 37%. Therefore, a significant number of defects remain in the SRS when the lifecycle proceeds to subsequent phases of development.

The low performance of traditional requirements inspection methods can be attributed to the following causes. As the size and complexity of software systems increase, it is difficult for an inspector to verify a large number of definitions of functions and variables, and analyze system behaviors for different possible use scenarios.

To reduce the number of defects that propagate from the requirements phase to other lifecycle phases, it is important to have an effective and efficient method for software requirements inspection. In this paper, a requirements inspection method based on scenarios generated by model mutation (RIMSM) is proposed to help inspectors identify defects in the functional requirements of a safety-critical system.

The RIMSM method models software requirements using a High Level Extended Finite State Machine. A method that executes the HLEFSM model is defined. The method uncovers the behaviors and generates the outputs of the system for a given scenario. A scenario in this paper is defined as a specific use case in which the system operates. A defect can be identified when the behaviors or outputs of the system do not meet our expectations.

The success of identifying defects in an HLEFSM model requires that the model is executed for an adequate set of scenarios and all defects in the model are uncovered in the results of execution. To identify an adequate set of scenarios, we extend the concept of mutation testing to the requirements level and apply it to the HLEFSM model of the requirements. Traditional mutation testing is effective in identifying an adequate test set for the code level (Jia and Harman 2011). It follows a set of rules which are called mutation operators to systematically seed faults into the source code. Each seeded fault results in a new version of the program, which is called a mutant. The test set is then evaluated based on its capability of distinguishing the mutants from the original program. In this paper, twenty-one mutation operators are designed based on a taxonomy of defects defined for the requirements phase. Mutants of the HLEFSM model are generated using these operators. An algorithm is developed to identify scenarios that can kill the model mutants. The set of scenarios is considered to be adequate for detecting defects in the model when all model mutants generated are killed. The HLEFSM model is then executed for the scenarios generated. The results of execution are used to detect defects in the model.

The RIMSM method allows the inspectors to focus on the behaviors and outputs of the system in a given scenario each time. The performance of inspection can be improved by reducing the amount and complexity of the information that needs to be analyzed by the inspectors simultaneously.

An automation tool, RITSM (i.e. Requirements Inspection Tool based on Scenarios Generated by Model Mutation), is developed to help inspectors apply the RIMSM method. The RITSM tool takes inputs from the inspectors to construct the HLEFSM model of an SRS. The mutants of the model and the scenarios by which the model mutants can be killed are generated automatically. The RITSM tool then executes the HLEFSM model in the generated scenarios and lists the behaviors and the outputs of the system for each scenario. By inspecting the results of the RITSM tool, the defects in the HLEFSM model can be detected and then their location identified in the SRS.

A question that needs to be answered then is whether the RIMSM method helps improve the performance of SRS inspection. An experiment is designed and conducted to evaluate the performance and usability of the RIMSM method by comparing the RIMSM method to the Checklist-Based Reading method.

The paper is organized as follows. Section 2 reviews previous research on requirements inspection techniques and mutation testing. Section 3 provides an overview of the RIMSM method. Based on the detailed steps of the RIMSM method introduced in Section 3, the HLEFSM is introduced in Section 4. Section 5 introduces the method to execute an HLEFSM model using which the behaviors and outputs of a system in a given scenario can be displayed. In Section 6, the mutation operators defined for the HLEFSM are introduced. An algorithm is developed in Section 7 to identify an adequate set of scenarios killing all model mutants. The RITSM tool is introduced in Section 8. In Section 9, an experiment is designed and conducted to determine the level of performance and usability of the RIMSM method.

2 Related work

A requirements inspection method, RIMSM, is proposed in this paper. This research is motivated by the insufficient effectiveness of the traditional requirements inspection techniques such as Ad Hoc Reading, Checklist-Based Reading, and Perspective Based Reading, as discussed in the introduction. In this section, related research on requirements inspection is reviewed in detail. In the RIMSM method, mutation testing is extended to the requirements phase and applied to an HLEFSM model to help generate a sufficient set of scenarios for the detection of defects. Relevant studies on mutation testing are also discussed.

The Ad Hoc Reading method is a nonsystematic approach to inspection of the requirements. In the method, no description is given as to how the document should be read. All inspectors are assigned the same general responsibilities to inspect an SRS. The performance of the inspectors thus highly depends on their capability (Porter et al. 2002).

To improve the performance of these inspections, the Checklist-Based Reading (CBR) technique provides the inspectors with a checklist that is expressed in the form of questions or statements in order to search for a specific type of defect (Staron et al. 2005). Inspectors read the document while answering a list of yes or no questions to detect defects (Lanubile and Visaggio 2000). The CBR technique is also nonsystematic (He and Carver 2006) because it does not provide instructions on how the inspection should be performed. In the CBR method,

every inspector is responsible for the whole inspection process. Since all inspectors share the same checklist, the overlap in their work leads to a waste of inspectors' effort (Yang 2007).

To better utilize inspectors' effort, the Perspective-Based-Reading technique allows the inspectors to read an SRS from different perspectives (e.g. user, designer, tester). The Perspective-Based-Reading technique operates under the premise that a different subset of the information contained in the requirements is more or less important for the different users of the document. Inspectors taking different perspectives are assumed to find different defects. For each perspective, an inspection procedure is defined. Perspective-Based-Reading is considered to be systematic since it explicitly identifies the different uses for the requirements and gives the inspectors a definite procedure for verifying whether those uses are achievable (Laitenberger 2002).

To further improve the performance of requirements inspection, new SRS inspection techniques have been recently proposed. Recent research on SRS inspection is introduced briefly below. A. Alshazly et al. (2014) proposed a combined-reading technique that separates the SRS into different parts based on the purpose of each part. In each part, defects are searched using a specifically designed checklist. However, there is insufficient evidence to conclude whether the method exhibits higher effectiveness and efficiency. F Dalpiaz et al. (2019) used the Natural Language Processing technique to detect terminological ambiguity in user requirements. The technique showed higher efficiency and equal effectiveness compared to the Ad Hoc Reading Technique. The result is limited to the detection of ambiguities only. Ali et al. (2018) proposed a method to qualify the SRS by standardizing the process of requirements generation. The standard SRS can then be inspected by a third party. A Total Quality Score is used to quantify the quality of the SRS generated. No experimental data are available for the method proposed. These methods improve either the effectiveness or the efficiency of the inspection process. A systematic method that demonstrates an overall performance improvement is still absent.

In this paper, a requirements inspection method RIMSM is proposed. Since traditional mutation testing is effective in identifying an adequate test set in the code level (Jia and Harman 2011), the concept of mutation testing is extended to the requirements phase and applied to an HLEFSM model in order to identify an adequate set of scenarios for detecting defects in the SRS. Previous studies on mutation testing are discussed below.

Mutation Testing is a fault-based software testing technique that has been widely studied for about four decades (Jia and Harman 2011). The purpose of mutation testing is to generate an effective test set, which can detect faults, by repeatedly improving the test cases. The concept of Mutation Testing was proposed in 1971 by Richard Lipton (1971). The birth of the field can be traced back to the late 1970s by Demillo et al. (1978), and Hamlet (1977). Mutation testing has been applied to test the program source code implemented by different languages such as Fortran (Budd et al. 1978; King and Jefferson Offutt 1991), Java (Kim et al. 1999; Ma et al. 2005), C (Agrawal et al. 1989), etc.

Mutation testing has also been applied at the software design level which is often referred to as "specification mutation". Budd and Gopal (1985) proposed specification mutation in 1985 to generate test cases for code testing. The specification is modeled in the predicate calculus (PC) form. However, the predicate calculus form is not able to model complex systems. Fabbri et al. (Pinto Ferraz Fabbri et al. 1994; Fabbri et al. 1999) investigated a mutation analysis method for specification in the format of the Finite State Machine (FSM) and Statecharts. The mutants were used to evaluate the effectiveness of the pre-defined test sequences. A tool Proteum/FSM was developed to automate the mutant generation process. Okun (2004), Lorber

et al. (2018) and Zhang et al. (2019) applied mutation testing on specifications modeled by Symbolic Model Verifier (SMV). The properties that the system needs to satisfy were specified using the computation tree logic (CTL). Model checking technique was used to generate execution paths that kill the mutants. The execution paths generated are used as test cases for the code level. Sugeta et al. (2004) focused on specifications written in Specification and Description Language (SDL). Mutation testing was used to assess the test cases generated by model checking techniques. Hierons (Hierons and Merayo 2007; Hierons and Merayo 2009) investigated the application of Mutation Testing to Probabilistic Finite State Machines (PFSMs). The method was proposed to evaluate test sequences generated by other testing methods for PFSM. Zhang et al. (2016) focused on requirements specified by Restricted Use Case Modeling (RUCM). Mutation testing was used to evaluate other test case generation methods. Sullivan and Feinn (2012) studied mutation testing for specifications written in the Alloy language. The mutants generated are used to evaluate the two methods of test case generation that they proposed.

In traditional specification mutation, the specification is described in the form of a modeling language, such as PC, FSM, PFSM, etc. The specification is assumed to be defect-free. The mutation operators are designed based on the defects that may appear in the model. The mutants are used to evaluate the pre-defined test cases for the model or generate test cases for the code level.

The RIMSM method in this paper starts with an SRS in natural language. The SRS is assumed to contain defects. Mutation operators are designed based on the potential defects that appear in the SRS. Since a defect in an SRS may cause multiple errors in its model, the mutation testing in this paper is a high order mutation for the HLEFSM. The mutants are used as inputs to an algorithm to identify an adequate set of scenarios for executing the HLEFSM model. The results of execution are used to detect defects in the SRS.

compares the model mutation in this paper with previous studies. The first row of the table displays the name of the authors. The second row shows the form of the specification focused in each study. The third row gives the name of the model used to represent the specification. The fourth row introduces whether the specification is assumed to be correct in each paper. The fifth row introduces the number of mutation operators developed in each study and the sixth row introduces the coverage of the mutation operators developed. The next three rows compare the goals of each study. The last row specifies whether an automation tool was developed for the method proposed (Table 1).

In this paper, the RIMSM method models the SRS using the HLEFSM which was proposed in (Li et al. 2016; Li and Gupta 2013; Li and Smidts 2017). In their research, the HLEFSM is used to analyze the branches of a system and predict the reliability of the system in the requirements phase. Defects that appear in the requirements phase were also studied in (Li et al. 2016; Li and Gupta 2013; Li and Smidts 2017). The details of the HLEFSM and the taxonomy of defects in the requirements phase are introduced in section 4.

The original contributions in this paper consist of the following. First, a method that executes the HLEFSM model is defined. Second, twenty-one mutation operators are designed based on the taxonomy of defects defined for the requirements phase. Third, an algorithm is developed to identify scenarios that can kill the model mutants generated using the mutation operators. Fourth, an automation tool, RITSM, is developed to support the application of the RIMSM method. Last, an experiment is designed and conducted to validate the RIMSM method.

Table 1 Comparison of Specification Mutation Papers

Author	Budd and Gopal 1985	Pinto Ferraz Fabbri et al. 1994	Okun 2004	Sugeta et al. 2004	Hierons and Merayo 2009	Sullivan and Feim 2012	Zhang et al. 2016	Lorber et al. 2018	Zhang 2018	Li 2020
Specification expressed as	PC	FSM	SMV	SDL	PFSM	Alloy	RUCM	SMV	SMV	SRS in Natural language
Model	PC	FSM	SMV	SDL	PFSM	Alloy	RUCM	SMV	SMV	HLEFSM
The specification is assumed to be correct or not	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
Coverage of mutation operators	<i>Predicates in the PC model</i>	<i>Transition logic of the FSM model</i>	<i>Predicates of SMV model</i>	<i>Elements of SDL model</i>	<i>Transitions of the PFSM model</i>	<i>Elements of Alloy model</i>	<i>Elements of use case models</i>	<i>Elements of SMV model</i>	<i>Predicates of SMV model</i>	<i>Elements of the HLEFSM model</i>
# Mutation operators	5	9	8	49	4	9	25	7	4	21
Goal Identify scenarios to verify the SRS		✓		✓	✓	✓	✓			✓
Estimate the effectiveness of pre-defined test cases for the specification model										
Generate test cases for the code level	✓		✓					✓	✓	
Automation Tool		✓	✓					✓	✓	✓

3 Overview of the RISM method

The detailed steps of the RISM method include: 1) construction of the model of the SRS, 2) generation of model mutants, 3) generation of scenarios under which the mutants can be distinguished, 4) execution of the model in generated scenarios, 5) identification of defects by inspecting the results of execution.

The flowchart of the RISM method is shown in Fig. 1. The RISM method starts with the SRS of a system. The rectangles are the outputs of each step.

The model of the SRS is constructed in step 1. In this paper, a High Level Extended Finite State Machine (HLEFSM) is used to model software requirements. In step 2, the mutants of the model are generated based on mutation operators defined for the HLEFSM model. Step 3 generates scenarios that kill all non-equivalent mutants from step 2. The set of scenarios are considered adequate to detect defects in the model. In step 4, the model is executed for the scenarios generated in step 3. The results of execution uncover the behaviors and display the outputs of the system for each scenario. The results of execution are reviewed by inspectors in step 5. If the results of execution in a scenario do not meet the expectations of the inspectors, a defect can be identified in the model and the SRS.

The tool RITSM automates step 2, step 3, and step 4. The details of the RISM method are introduced in the following sections.

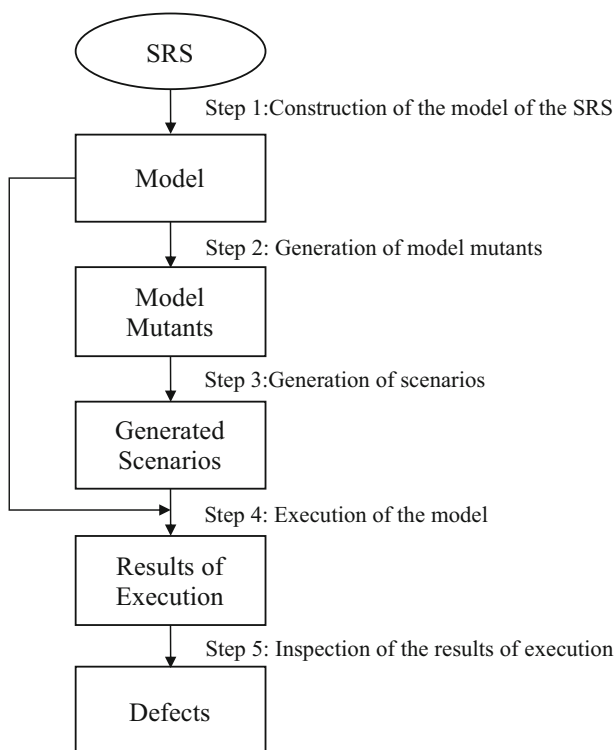


Fig. 1 Overview of the RISM method

4 Modeling a system in the requirements phase

This section introduces the first step of the RIMSM method, i.e. modeling a system based on its Software Requirements Specification (SRS). The content and structure of an SRS are first introduced. The High level extended finite state machine (HLEFSM) is then discussed to model the SRS. The steps to construct an HLEFSM model are then introduced. At the end of this section, the taxonomy of defects defined for the SRS is discussed. The HLEFSM model and the taxonomy of defects introduced in this section were proposed in (Li et al. 2016; Li and Gupta 2013; Li and Smidts 2017).

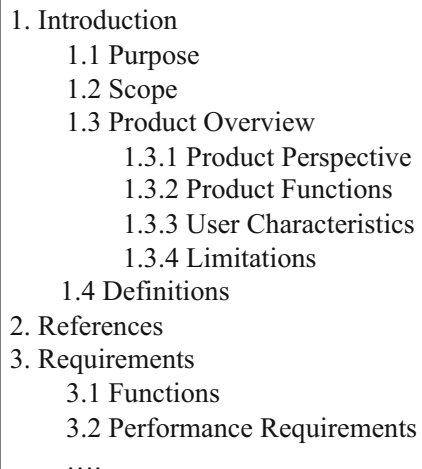
4.1 Software requirements specification

An SRS captures the software requirements that will drive the design and design constraints to be considered or observed (Li et al. 2018). The IEEE 29148:2018 (IEEE 2018) standard integrates a series of IEEE standards related to requirements engineering (e.t. ISO/IEC/IEEE 12207:2008, ISO/IEC/IEEE 15288:2008, ISO/IEC/IEEE 15289:2011, ISO/IEC TR 19759, IEEE Std 830, IEEE Std 1233, IEEE Std 1362, ISO/IEC TR 24748–1, and ISO/IEC/IEEE 24765). The IEEE 29148:2018 standard defines the processes of the requirements engineering activity and formulates requirements for software requirements documentation. An example SRS outline from the IEEE 29148:2018 standard is provided in Fig. 2.

This paper focuses on requirements related to software functions to ensure that the functionality of the software under implementation is correctly described in its SRS. As defined in IEEE 29148:2018, the information related to software functions in a normative SRS includes:

- **Purpose and Scope:** the purpose and scope of the SRS (e.g. section 1.1 and 1.2 in Fig. 2);
- **Product Overview:** a summary of the major functions that the software will perform (e.g. section 1.3 in Fig. 2). Use cases, user stories, and scenarios are also used to describe product functions.

Fig. 2 A typical SRS outline based on the IEEE 29148:2018 standard

- 
1. Introduction
 - 1.1 Purpose
 - 1.2 Scope
 - 1.3 Product Overview
 - 1.3.1 Product Perspective
 - 1.3.2 Product Functions
 - 1.3.3 User Characteristics
 - 1.3.4 Limitations
 - 1.4 Definitions
 2. References
 3. Requirements
 - 3.1 Functions
 - 3.2 Performance Requirements
 -

- **Specific Functions:** the fundamental actions that have to take place in the software in accepting and processing the inputs and in processing and generating the outputs (e.g. section 3.1 in Fig. 2).

A specific function consists of the following information:

- *Function name.*
- *Functionality: this information describes the goal of the function.*
- *Variables: variables include input variables, output variables, and intermediate variables of the function. A variable is a tuple with three elements:*
 - *Variable name: the name of the variable.*
 - *Type: can be integer, decimal, string, or Boolean.*
 - *Range: the range of this input.*
- *Sub-Functions: a sub-function forms part of the function.*
- *Function logic: this information describes how the function achieves its functionality. The function logic consists of four logic components: function calls, variable assignments, predicates, and loop statements.*

The functions defined in an SRS can be represented in a tree structure (see Fig. 3). A system itself is defined as a level-0 function which is implemented by the major functions (level 1 function) specified in section 1.3 of Fig. 2. A level 1 function can be specified by its sub-functions which are defined as level 2 functions. For example, the function “Lv2 fun1.1” in Fig. 3 is a level-2 function used to specify the level-1 function “Lv1 fun1”.

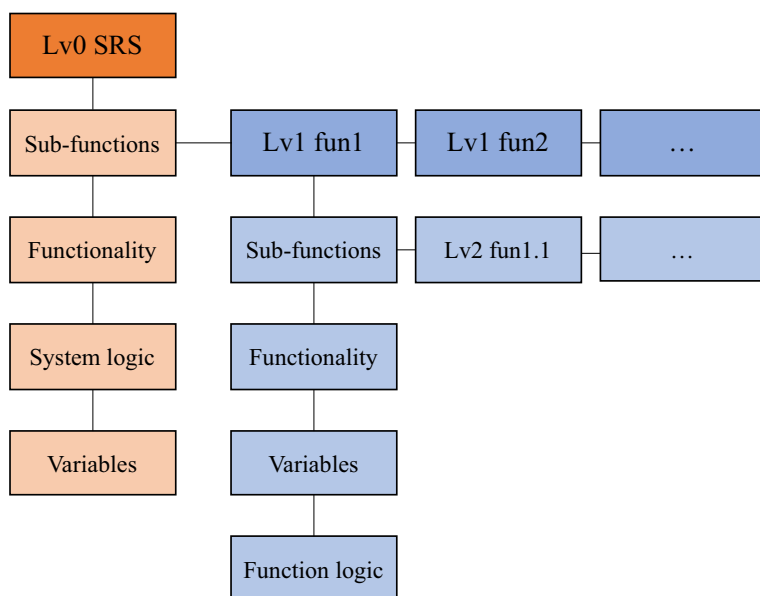


Fig. 3 SRS structure

4.2 HLEFSM

In the study of Li. et al. (2016), a High-Level Extended Finite State Machine (HLEFSM) was defined to model software requirements. In this paper, the HLEFSM is used for the following reasons. First, the HLEFSM models software requirements hierarchically. The tree structure of the functions in an SRS can be modeled directly without extra effort. Second, a component of an HLEFSM model can be easily mapped to its definition in the SRS. Once a defect is detected in the HLEFSM model, its location in the SRS can be easily identified.

The HLEFSM in this paper is a nonuple $(\Sigma, \Gamma, IV, S, T, F, SL, P, A)$ which is defined as follows:

- Σ is the set of input variables of the software. These variables cross the boundary of the software application.
- Γ is the set of output variables of the software. These variables cross the boundary of the software application.
- IV is the set of internal variables defined and used within the software application boundary.
- S is a finite, non-empty set of states. A state corresponds to the condition of the software application.
- T is the set of transitions. A transition is an event that causes a state change of the system. A transition connects two states of the system and indicates the direction of the state change.
- F is the set of functions that are defined in the system. A function is specified by the function name, functionality, and function logic. A function instance consists of two states, the transition between the two states and the relation of outputs to inputs during the transition. The transition in a function can be a combination of transitions in T .
- SL is the set of sequential links. A sequential link connects two adjacent logic components to represent the sequential execution of the logic components.
- P is the set of predicates. The logical value of the predicates is attached to the related sequential link.
- A is the set of assignments. An assignment consists of two states and the transition between the two states.

An example of HLEFSM is given in Fig. 4. Let us assume that a system takes temperature as its input. The system first performs an initialization function to initialize the system. If the temperature is greater than 45 C, the system performs the function open_valve to open a valve. Then the system stops and outputs a control signal.

The HLEFSM is constructed based on the requirements related to functions. The HLEFSM model of the example above is defined as below where Fun0 is the level 0 function and Fun1–3 are three level 1 functions. The logic of the system is shown in Fig. 4.

- $\Sigma = \{\text{Temp}\}$
- $\Gamma = \{\text{control_signal}\}$
- $IV = \emptyset$
- $S = \{\text{System not Initialized, System Initialized, Valve Closed, Valve Opened, System not Stopped, System Stopped}\}$.
- $T = \{\text{Initialize_System, Open_Valve, Stop_System}\}$.

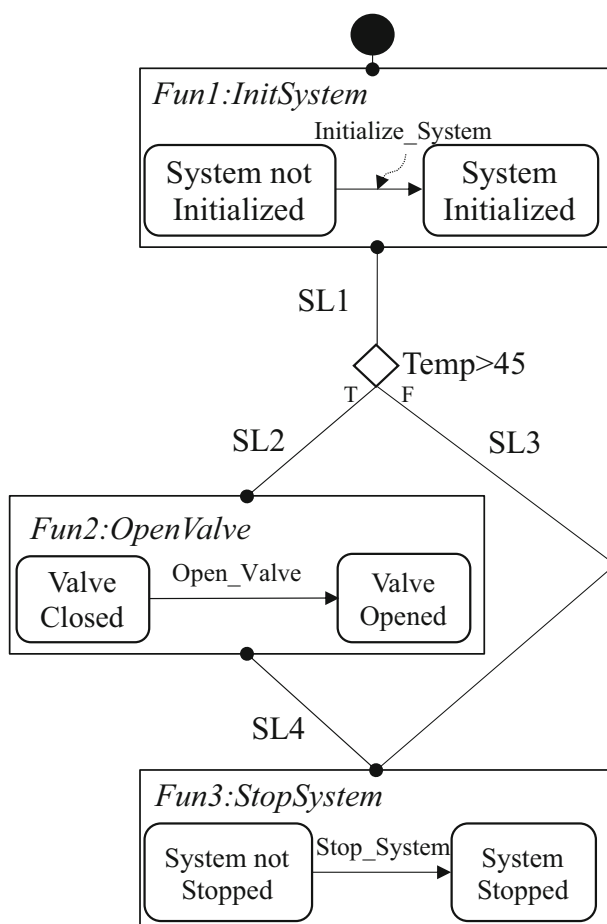


Fig. 4 HLEFSM model

- $F = \{\text{Fun0:SRS}, \text{Fun1:InitSystem}, \text{Fun2: OpenValve}, \text{Fun3: StopSystem}\}.$
- $SL = \{SL1, SL2, SL3, SL4\}.$
- $P = \{\text{Temp}>45\}$
- $A = \emptyset$

4.3 Construction of an HLEFSM model based on the SRS

Constructing the HLEFSM model of an SRS follows the steps in Table 2. Each step is detailed by the substeps.

4.4 Defects in SRS

Defects that appear in the requirements phase were studied by Li et al. based on the content of the SRS (Li, Mutha, and Smidts 2016; Li and Gupta 2013; Li and Smidts 2017). In Table 3,

Table 2 Constructing the HLEFSM model of an SRS

Step	Description
1	Define the set of functions
1.1	Define the level 0 function which is the SRS itself
1.2	For the functions that are already defined, define their sub-functions in the lower level based on the specification of each function in the SRS.
1.3	Repeat Step 1.2 until no functions can be further defined
2	Define the set of input variables, output variables, and intermediate variables
2.1	Identify and define the input variables, output variables, and intermediate variables based on the specification of each function
3	Define the set of predicates
3.1	Identify predicates in the logic of the functions.
4	Define the set of assignments
4.1	Identify assignments in the logic of the functions.
5	Define the set of transitions
5.1	Define a transition for each function instance in the logic of the functions
5.2	Define a transition for each assignment in the logic of the functions
6	Define the set of states
6.1	Identify two system states for each transition in the logic of the functions
7	Define the set of sequential links
7.1	Connect the logic components based on the system logic. Each connection is a sequential link.

twenty-one types of defects commonly appearing in the SRS are given in the second column. The defects are grouped into three categories as shown in the first column of Table 3.

It should be noted that the defects in an HLEFSM model include the defects that originate from the SRS as well as the defects that are introduced during the modeling process. Once a defect is identified in the model, its location in the SRS is easily identifiable by mapping the defective component of the model to its definition in the SRS. If a defect identified in the

Table 3 Defects in SRS

Category	Defect name
Defects related to the definition of functions	1. Missing (definition of) function 2. Extra (definition of) function 3. Incorrect/ambiguous function name 4. Function with Incorrect functionality
Defects related to the definition of variables	5. Missing variable 6. Extra variable 7. Incorrect/ambiguous variable name 8. Variable with incorrect type 9. Variable with incorrect range
Defects related to function logic	10. Missing instance of function 11. Extra instance of function 12. Incorrect/ambiguous function call. 13. Missing assignment 14. Extra assignment 15. Incorrect/ambiguous assignment 16. Missing predicate 17. Extra predicate 18. Incorrect/ambiguous predicate 19. Missing Loop 20. Extra Loop 21. Incorrect/ambiguous loop

model cannot be found in the SRS, it is considered as a defect introduced in the construction of the model. Defects introduced during the construction of the model will not be recorded as SRS defects.

5 The execution method of an HLEFSM model

The method defined to execute the HLEFSM model is discussed in this section. The execution aims to uncover the behaviors and generate the outputs of the system in a given scenario. A scenario is a use case that is specified with a set of inputs to the HLEFSM model. By executing the model under different scenarios, the behaviors and outputs of the system can be analyzed using the results of execution. The defects within an HLEFSM model can be identified if the behaviors or outputs of the system do not meet our expectations.

The functions in an SRS follow the tree structure shown in Fig. 3. Each function is detailed by its name, functionality, sub-functions, variables, and logic. The variables in each function are classified into input, output, and intermediate variables. Each variable is detailed by its name, type, and range. The root of the tree is the Lv0 function which is the SRS itself. In this section, a method to execute the HLEFSM model is developed. The method follows three steps which are 1) execute the definitions of functions, 2) execute the definitions of variables, 3) execute the logic of functions.

5.1 Step 1- execution of function definitions

In the first step, the functions at each level are traversed. The names and functionalities of the functions are outputted into a string. The format of the string is defined in Fig. 5. In the first line of Fig. 5, the level 0 function “SRS” is displayed. The string “[SRS, fun1, functionality1]” indicates that the level 0 function “SRS” has a level 1 sub-function whose name is “fun1” and whose functionality is “functionality1”. The level 1 functions are displayed in the second line.

In this step, the name and functionality of each function are listed. The result of execution is examined manually by the inspectors to identify the defects related to the definition of functions. Listing the functions hierarchically helps the inspectors better analyze the relations between the system functions and identify related defects.

5.2 Step 2- execution of variable definitions

In the second step, the variables defined in each function are outputted into a string. By traversing variables in each function, the input, output, and intermediate variables are outputted separately as shown in Fig. 6.

The square bracket shows the target function and variable class (inputs, outputs, or intermediate variables). The angle brackets list all variables in that class with the name, type,

```
<lv0>[SRS]
<lv1>[SRS, fun1, functionality1] [SRS, fun2, functionality2]...
<lv2>[fun1, fun1.1, functionality1.1] [fun2, fun2.1, functionality1.2]...
...
```

Fig. 5 Examination of Function definitions

[Target Function1, Inputs]
<Input1, Type, Range, Boundary><Input2, ...>
[Target Function1, Outputs]
<Output1, Type, Range, Boundary><Output2, ...>
[Target Function1, Variables]
<Variable1, Type, Range, Boundary><Variable2, ...>

Fig. 6 Examination of variable definitions

range, and application boundary of each variable. The outputs of this step can be used to identify defects related to the definition of variables.

5.3 Step 3-execution of function logic

In the third step, the logic underlying a function is executed for a given scenario.

The execution in step 3 starts from the logic of the level 0 function. When a function instance is encountered during execution, the model goes into the logic of the instantiation of that function. A table of variables is maintained during execution to update the value of all variables when an assignment is executed. The function instances, predicates and loops that the system encounters during execution are outputted as a string. The format of the output string is defined in Table 4.

The instances of functions that the system encounters during execution are outputted in the square brackets. The logic inside a function instance is displayed in the brace following the function instance.

A predicate and its evaluation are placed in an angle bracket followed by parentheses that contain the logic within the predicate block.

A loop is executed as a sequence of predicates. A loop is placed in a brace. The condition of the loop is placed in an angle bracket. The executed logic within the loop block for each iteration is placed in a sequence of parentheses following the angle bracket. In this paper, we assume there is no infinite loop.

An example of the output string is displayed below.

$$[SRS, ([fun2, ()] < Predicate1, true, ([fun1, ([fun1.1, ()]]) > \{ < Loop1 >, ([fun3, ()]), ([fun3, ()]) \})]$$

In the example the system first encounters the level 0 function “[SRS]” and goes into its logic in the following parentheses. The system executes function instance “*fun2*” which has no detailed logic. “*Predicate1*” is then evaluated to be true and the system executes the logic inside the predicate block. As the system traverses the predicate block of “*Predicate1*” the system meets function instance “*fun1*” who calls its sub-function “*fun1.1*” in its logic. Then the system iterates “*Loop1*” twice. In both iterations of “*Loop1*” the system executes function instance “*fun3*” which has no detailed logic. The loop condition is evaluated to be false in the third iteration

The results of execution in this step uncover the behaviors and generate the outputs of the system in a given scenario for identifying defects related to the logic of the functions.

5.4 Example for illustration

This section illustrates the execution method using the HLEFSM example in Fig. 4. Given a scenario $Temp = 50$, the results of execution of the example system are discussed below.

In step 1, the definitions of functions are executed. The system has one level 0 function and three level 1 functions. The outputs of step 1 are displayed below.

```
< lv0 > [SRS]
< lv1 > [SRS, InitSystem, initialize the system]
        [SRS, OpenValve, open the valve]
        [SRS, StopSystem, stop the system]
```

In step 2, the definitions of the variables are executed. Only two variables are defined in the system level (i.e. in the level 0 function). The range and type of the two variables are not specified. The outputs of step 2 are given below.

```
[SRS, Inputs]
    < Temp, None, None >
[SRS, Outputs]
    < Control Signal, None, None >
```

In step 3, the logic of each function is executed. In the example system, only the logic of the level 0 function is specified. The results of execution are shown below.

```
[SRS, ([InitSystem, ()] < Temp > 45, true, ([Open_Valve, ()]) > [Stop_System, ()])]
```

6 Mutation Testing Extension to the Requirements Phase

In this section, we introduce the extension of mutation testing. Traditional Mutation Testing is based on two hypotheses (Hamlet 1977). One is the competent programmer hypothesis which states that most software faults introduced by experienced programmers are due to small syntactic errors. The other hypothesis is called the coupling effect which asserts that simple faults can cascade or couple to form other emergent faults.

Table 4 Output String Format

Component	Format	Description
Function Instance	[Function instance, (logic)]	A function instance is in the square brackets. The Logic of the function is in the parentheses following the function instance. If the function instance has no detailed logic, its logic is empty.
Predicate	<Predicate condition, Evaluation, (logic)>	The predicate condition and its evaluation are placed in the angle brackets. The logic inside the predicate block is in the parentheses following the condition.
Loop	{<Loop condition>, (logic of the 1st iteration), (logic of the 2nd iteration),...}	The condition of the loop is placed at the beginning inside the brace. For each iteration, the executed logic within the loop block is placed in a sequence of parentheses.

In this paper, the hypotheses of mutation testing are considered to be valid in the requirements phase as well. The reason is that SRS is a high-level description of a software system. The SRS is developed by people who are competent in requirements engineering. The defects that appear in an SRS are also small syntactic errors (see defects in Table 3). Those small errors can cascade or couple to form other serious defects in the requirements phase.

In this section, twenty-one mutation operators are defined below to mimic the 21 types of defects that could be found in an SRS as shown in Table 3. For each mutation operator, detailed rules following which mutants can be generated are developed based on the corresponding defect being modeled.

MF (Missing function): The mutation operator MF is defined based on the defect missing (definition of) function. An MF mutant can be generated by removing the entire definition of a function from an HLEFSM model. There is one mutant corresponding to each function definition.

EF (Extra function): The mutation operator EF is defined based on the defect extra (definition of) function. An EF mutant can be generated by duplicating the definition of an existing function. There is one mutant corresponding to each function definition.

IAFN (Incorrect/ambiguous function name): The mutation operator IAFN is defined based on the defect incorrect/ambiguous function name. An IAFN mutant can be generated by modifying the name of a function randomly. There is one mutant corresponding to each function definition.

FIF (Function with incorrect functionality): The mutation operator FIF is defined based on the defect function with incorrect functionality. An FIF mutant can be generated by modifying the functionality of a function randomly. There is one mutant corresponding to each function definition.

MV (Missing variable): The mutation operator MV is defined based on the missing variable defect. An MV mutant can be generated by removing the definition of a variable from a function.

EV (Extra variable) The mutation operator EV is defined based on the defect extra variable. An EV mutant can be generated by duplicating the definition of a variable in a function. There is one mutant corresponding to the definition of each variable.

IAVN (Incorrect/ambiguous variable name): The mutation operator IAVN is defined based on the defect incorrect/ambiguous variable name. An IAVN mutant can be generated by modifying the name of a variable randomly. There is one mutant corresponding to the definition of each variable.

VIT (Variable with incorrect type): The mutation operator VIT is defined based on the defect variable with incorrect type. A VIT mutant can be generated by replacing the type of a variable with another type. If a system contains T types of variables, there are $T - 1$ mutants corresponding to the definition of each variable. In this paper, we have $T = 4$ since the types of variables are integer, decimal, string, and Boolean.

VIR (Variable with incorrect range): The mutation operator VIR is defined based on the defect variable with incorrect range. A VIR mutant can be generated by replacing the constants in the range of a variable with other randomly selected constants. There is one mutant corresponding to the definition of each variable.

MIF (Missing instance of function): The mutation operator MIF is defined based on the defect missing instance of function. An MIF mutant can be generated by removing one

instance of a function from the logic of the system. There is one mutant corresponding to each function instance in the system logic.

EIF (Extra instance of function): The mutation operator EIF is defined based on the defect extra instance of function. An EIF mutant can be generated by inserting an instance of a function into a sequential link in the system logic. The inserted instance is randomly selected from the existing function instances of the model. There is one mutant corresponding to each sequential link in the HLEFSM model.

IAFC (Incorrect/ambiguous function call): The mutation operator IAFC is defined based on the defect incorrect/ambiguous function call. An IAFC mutant can be generated in two ways: 1) replacing a function instance with another function instance; 2) adding a random suffix to the name of the instance of a function. Two mutants can be generated for each function instance in the system logic.

MA (Missing assignment): The mutation operator MA is defined based on the defect missing assignment. An MA mutant can be generated by removing an assignment from the system logic. One mutant can be generated for each assignment in the system logic.

EA (Extra assignment): The mutation operator EA is defined based on the defect extra assignment. An EA mutant can be generated by inserting an assignment into a sequential link in the logic of a function. The inserted assignment shall assign a variable defined in the function with a constant randomly selected from the range of the variable. Let us assume that the assignment is inserted into the logic of a function with V_i variables defined. There are V_i mutants corresponding to each sequential link in the logic of the function.

IAA (Incorrect/ambiguous assignment): The mutation operator IAA is defined based on the defect incorrect/ambiguous assignment. Let us consider A_i an assignment in the logic of a function. There are V_i variables defined in the function. The assignment A_i consists of o_i arithmetic operators, v_i variables and c_i constants. An IAA mutant can be generated by modifying the assignment A_i in the following ways.

Replacing an arithmetic operator $\{+, -, *, /, ^\}$ in A_i with another arithmetic operator. There are four mutants corresponding to each operator.

Replacing a variable in A_i with another variable. There are $V_i - 1$ mutants corresponding to each variable.

Replacing a variable in A_i with a constant. There is one mutant corresponding to each variable.

Replacing a constant with a variable. There are $V_i - 1$ mutants corresponding to each constant.

Replacing a constant in an assignment with another constant. There is one mutant corresponding to each constant in A_i .

For each assignment A_i , the number of IAA mutants that can be generated is $4o_i + V_i(v_i + c_i)$.

MP (Missing predicate): The mutation operator MP is defined based on the defect missing predicate. An MP mutant can be generated by removing a predicate from the logic of a function. The logic inside the predicate block stays at the location of the removed predicate. There is one mutant corresponding to each predicate in the HLEFSM model.

EP (Extra predicate): The mutation operator EP is defined based on the defect extra predicate. An EP mutant can be generated by introducing a predicate into a

sequential link in the logic of a function. The logic components (i.e. assignment, function instance, loop, or predicate) following the sequential link are embedded in the predicate block. The condition expression of the inserted predicate can be either the Boolean constant “False” or the condition expression of another predicate that is randomly selected from the predicates in the same function as the target predicate. Two mutants are corresponding to each sequential link in the HLEFSM model.

IAP (Incorrect/ambiguous predicate): The mutation operator IAP is defined based on the defect incorrect/ambiguous predicate. Let us define that P_i is a predicate in the logic of a function. There are V_i variables defined in the function. The condition expression of the predicate P_i consists of o_i arithmetic operators, v_i variables, c_i constants, r_i relational operator (i.e. \leq , $<$, \geq , $>$, $=$, $!=$) and l_i logical connectors (i.e. and, or). An IAP mutant can be generated by manipulating the condition expression of P_i in the following ways. Replacing a relational operator with another relational operator. Five mutants are corresponding to each relational operator.

Replacing a logical connector (i.e. and, or) in the condition expression with another logical operator. There is one mutant corresponding to each logical operator.

Replacing an arithmetic operator $\{+, -, *, /, ^\}$ with another arithmetic operator. Four mutants are corresponding to each arithmetic operator.

Replacing a variable with another variable. There are $V_i - 1$ mutants corresponding to each variable.

Replacing a variable with a constant. There is one mutant corresponding to each variable.

Replacing a constant with a variable. There are $V_i - 1$ mutants corresponding to each constant.

Replacing a constant in an assignment with another constant. There is one mutant corresponding to each constant.

For each predicate P_i , the number of mutants is $5r_i + l_i + 4o_i + V_i(v_i + c_i)$

ML, EL, and IAL: The mutation operators ML, EL, and IAL are defined based on the defects missing loop, extra loop, and incorrect/ambiguous loop. A mutant can be generated following the same rule as defined for MP, EP, and IAP.

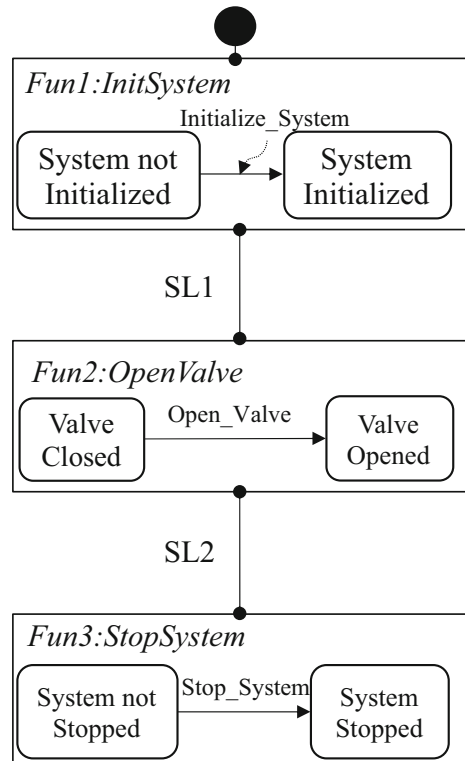
Mutants of the HLEFSM model are generated using these operators. A set of scenarios that kill all non-equivalent mutants are considered to be adequate to execute the HLEFSM model.

An example is given below to illustrate the process used to generate an MP mutant of the model in Fig. 4. The example contains only one predicate. A mutant can be generated by removing the predicate “Temp>45” from the logic of the system. The MP mutant generated is displayed in Fig. 7.

7 Generation of scenarios to kill mutants

In traditional mutation testing, the test cases are either selected by the testers manually or generated randomly to kill the mutants of the source code. In this section, an algorithm is introduced to identify an adequate set of scenarios that kill the mutants of an HLEFSM model.

Fig. 7 An MP mutant example



A scenario is a use case that is specified by defining a set of inputs to the system. The scenarios generated can be used to execute the HLEFSM model and identify defects in the requirements.

The following definitions are used in this section. An *execution path* is the sequence of logic components traversed by the model in a given scenario. A *logic expression* (LE) is a sequence of conditions. A logic expression describes execution paths that traverse a specific set of logic components. Solving a logic expression generates a set of scenarios. The model traverses one of the execution paths described by the logic expression if the model is executed under one of the solved scenarios.

The algorithm identifies scenarios following two main steps: 1) For each mutant, generate the logic expressions which represent the execution paths on which the mutant will be killed; 2) Solve all logic expressions to obtain the scenarios that kill all mutants.

To strongly kill a mutant, the logic expression needs to satisfy four criteria: 1) Necessity: the variables in the system satisfy their range, 2) Reachability: the execution paths reach the mutated component of the mutant, 3) Distinguishability: the execution paths create some differences between the mutant and the original SRS model immediately after the execution of the mutated component, 4) Sufficiency: the difference created propagates to the results of execution. The logic expressions which satisfy the four requirements are discussed below.

7.1 Logic expression: Necessity

Let us assume that a system has N variables. $VarR_i$ is the range of the i^{th} variable, Var_i . The logic expression for all inputs satisfying their ranges is:

$$VarR_{all} = (Var_1 \in VarR_1) \wedge (Var_2 \in VarR_2) \dots \wedge (Var_N \in VarR_N)$$

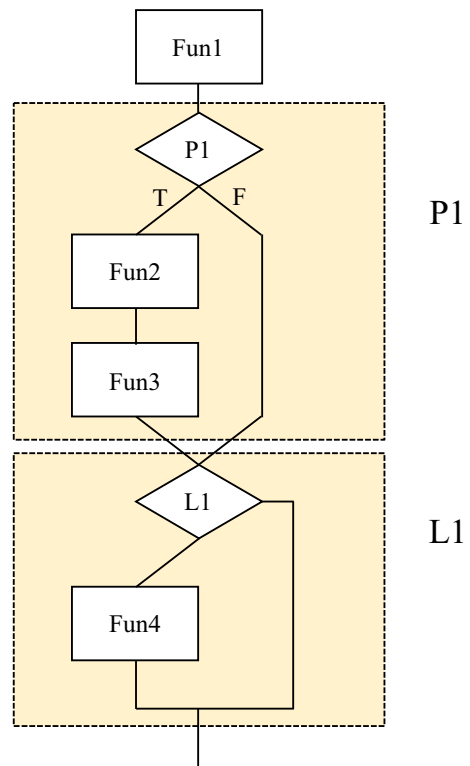
7.2 Logic expression: Reachability

Let us define that a *main component* in the system logic is the logic component that is always traversed during execution. A system example is shown in Fig. 8. The main components of the system are function instance “*Fun1*”, Predicate “*P1*” and Loop “*L1*”. The *main path* is defined as the sequence of main components traversed during execution.

Predicates and loops fork the main path into sub-paths. For example, the predicate “*P1*” in Fig. 8 has two branches. The loop “*L1*” has L branches, where L is the maximum number of iterations of the loop (each number of iterations is defined as a branch of a loop). There are $2 * L$ sub-paths in the system. One sub-path example is shown in Fig. 9 where *P1* is evaluated to be true and *L1* iterates once.

Given a set of inputs, the sub-path traversed by the system is defined as the execution path under that scenario. The logic expression of an execution path is the combination of the

Fig. 8 System Example



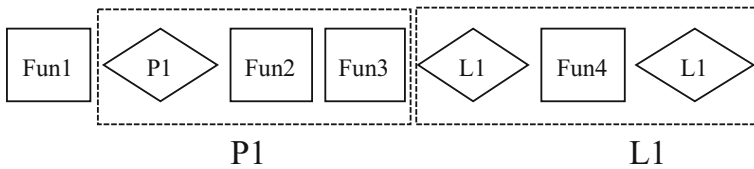


Fig. 9 Sub-Path Example

function instances, the assignments, the evaluation of each predicate traversed and the evaluation of the condition in each iteration of each loop traversed on the path. For example, the logic expression for the execution path in Fig. 9 is shown below. The inputs that satisfy the conditions defined by the logic expression will lead the system to traverse an expected execution path.

[Fun1 is called] and [P1 = true] and [Fun2 is called] and [Fun3 is called] and [L1_iter1 = true] and [Fun4 is called] and [L1_iter2 = false].

In the logic expression above, “L1_iter1” and “L1_iter2” are the evaluations of the loop predicate in the first and second iteration. “L1_iter1” and “L1_iter2” are different since the system executes “Fun4” between the evaluation of the loop condition in two iterations. An algorithm to generate the logic expressions of all execution paths in an HLEFSM model is provided in Fig. 10.

A recursive function is defined in Fig. 10 to generate the logic expressions of all execution paths of an SRS model. The function “*model.afterStmt(comp)*” at line 7 cuts the model at the location of a logic component “*comp*” and returns the unexecuted part of the model after “*comp*” for recursion. At line 9, if *comp* is a function instance, the logic expressions of the execution paths in the function instance are generated by calling the algorithm recursively. The generated logic expressions are returned to a set “*funInstance_LE_set*”. The function “*model.afterPred(pred, eval)*” at line 19 slices one branch of the predicate “*pred*” based on its evaluation “*eval*” and returns the rest of the model. The function “*model.loop_block(comp)*” at line 22 returns the logic components in the block of loop “*comp*”. The execution paths in the loop block are assigned to the variable “*LE_block*”. The function “*comp.loopLE(n, i, p)*” at line 26 returns a logic expression describing one path in the *i*th iteration of the loop. The variable “*Loop_LE_set*” at line 26 is a set of logic expressions describing the sub-paths of the loop.

Reachability requires the execution path to include the mutated component to kill a mutant. An execution path containing the mutated component can be divided into three parts: preceding path, location path, and posterior path.

A preceding path is a sequence of logic components executed before the main component that contains the mutated component. A mutant of the system in Fig. 8 is shown in Fig. 11. The mutant replaces the function instance “*Fun3*” with “*Fun5*”. Since the mutated component is located in predicate “*P1*”, there is only one preceding path which consists of function instance “*Fun1*”.

A location path is the sequence of logic components executed after the preceding path and before the mutated component. To reach the mutated component “*Fun5*”, the location path of the mutant in Fig. 11 consists of “*P1 = true*” and “*Fun2*” where “*P1*” needs to be evaluated to true.

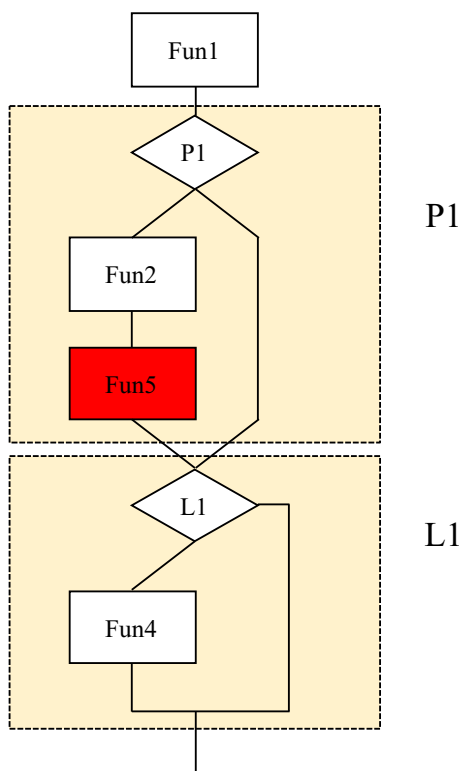
GenAllExePathLE ()	
Inputs: model: the HLEFSM model of the system or a subfunction	
Outputs: LE_set: A set of logic expressions for all execution paths	
1	LE is initialized as an empty string
2	LE_set, funInstance_LE_set, LE_block and Loop_LE_set are initialized as empty sets
3	comp ← model.get_next_comp()
4	if (comp is NULL)
5	return LE_set
6	if (comp is a function or a variable definition)
7	LE_set ← GenAllExePathLE(model.afterStmt(comp))
8	if (comp is a function instance)
9	funInstance_LE_set ← GenAllExePathLE (model.funInstance(comp))
10	for each funInstance_LE in funInstance_LE_set
11	for each subLE in GenAllExePathLE (model.afterStmt(comp))
12	LE_set.add(LE+funInstance_LE +subLE)
13	if (comp is an assignment)
14	LE ← LE + comp
15	for each subLE in GenAllExePathLE (model.afterStmt(comp))
16	LE_set.add(LE+subLE)
17	if (comp is a predicate)
18	LE ← LE + comp.condition(Eval)
19	for each subLE in GenAllExePathLE (model.afterPred(comp, Eval))
20	LE_set.add(LE+subLE)
21	if (comp is a loop)
22	LE_block ← GenAllExePathLE (model.loop_block (comp))
23	for each “n” where “n” is the number iterations of the loop :
24	for each iteration “i” of the loop:
25	for each path “p” in the loop block “LE_block” :
26	Loop_LE_set.add(loopLE(n, i, p))
27	for each loopLE in Loop_LE_set:
28	for each subLE in GenAllExePathLE (model.afterLoop(comp))
29	LE_set.add(LE + loopLE + subLE)
30	return LE_set

Fig. 10 Algorithm Generating Logic Expressions of All Execution Paths

A posterior path is a sequence of logic components executed after the location path. In Fig. 11, the posterior paths are the branches of “*LI*”. If the loop “*LI*” iterates twice given the system inputs, the posterior path consists of “*LI = true*”, “*Fun4*”, “*LI = false*” where the loop condition “*LI*” is evaluated to be true in the first iteration and false in the second iteration.

The algorithm that generates the preceding paths, location paths, and posterior paths for which the mutated component of a mutant can be traversed is shown in Fig. 12. The function *SeparatePathLE()* takes the logic expressions of all execution paths as inputs. If an execution path contains the mutated component, it is sliced into a preceding path, a location path, and a posterior path using the functions *slicePre()*, *sliceLoc()*, and *slicePost()*. The preceding path, location path, and posterior path are then added in a row of a three-column array *SeparatePathLE*.

Fig. 11 Mutant Example



To satisfy the reachability criterion, the execution path needs to traverse the mutated component of a mutant. Based on the method of executing an HLEFSM model, the definitions of functions and variables are traversed in any scenario. No specific logic expression is required to reach the mutated component of a mutant generated by the mutation operators

SeparatePathLE ()	
Inputs:	
LE_set: logic expressions of all execution paths	
mut_comp: mutated component	
Outputs:	
SeparatePathLE: a three-column array. Each row represents the logic expressions of the preceding path, location path and posterior path of an execution path which contains the mutated component	
1	for each le in LE_set:
2	if le contains mut_comp:
4	PrecedingLE \leftarrow slicePre(le, mut_comp)
5	LocationLE \leftarrow sliceLoc(le, mut_comp)
6	PosteriorLE \leftarrow slicePost(le, mut_comp)
7	SeparatePathLE.addRow(PrecedingLE, LocationLE, PosteriorLE)
8	return SeparatePathLE

Fig. 12 Algorithm Generating Preceding, Location, and Posterior Paths

related to the definitions of functions and variables, including MF, EF, IAFN, FIF, MV, EV, IAVN, VIT, and VIR.

For a mutant generated by the other mutation operators, the logic expressions of the qualified execution paths can be generated by combining the logic expression of a preceding path and a location path obtained using the algorithm in Fig. 12. Posterior paths will be used to fulfill the sufficiency criterion in later sections.

7.3 Logic expression: Distinguishability

Distinguishability requires the creation of some differences in system states between the mutant and the original SRS model immediately after the execution of the mutated component. In this section, distinguishability is discussed for mutants generated by each mutation operator.

Let us consider a missing function mutant from the mutation operator MF (Missing Function). Based on the method of execution defined, the system adds the mutated function to the output string in the first step of execution. Differences are created in the output string when the function is traversed. Since the distinguishability of an MF mutant is satisfied when the missing function is traversed, no specific logic expression of distinguishability is required. The mutants from the mutation operators modeling defects related to the definition of functions and variables do not need a specific logic expression of distinguishability for the same reason. The mutation operators include EF, IAFN, FIF, MV, EV, IAVN, VIT, and VIR.

A MIF mutant also creates differences in the output string when the mutated component is traversed. Let us assume a MIF mutant removes a function instance “*Fun_i*” from the original SRS model. When the location of “*Fun_i*” is traversed, the original SRS adds the string “*Fun_i*” to the results while the mutant does not add any string. Therefore, no specific logic expression is required to kill an MIF mutant since the distinguishability is satisfied when the mutated function instance is traversed. A mutant from the mutation operators IAFC and EIF does not require a specific logic expression of distinguishability for the same reason.

An IAA mutant creates differences in the table of variables maintained during execution. For example, an IAA mutant replaces the assignment “*var = assignment_expression*” with “*var = mutated_assignment_expression*”. To create differences after the execution of the mutated assignment, the variable “*var*” needs to be assigned a different value in the mutant. The logic expression for the distinguishability of an IAA mutant can be written as (1). The MA and EA mutation operators have the same logic expression as (1).

$$mutated_assignment_expression \neq assignment_expression \quad (1)$$

To create differences after executing the mutated component of an IAP mutant, one easy way is to traverse a different branch in the mutant after executing the mutated predicate. Let us denote the original predicate as P which is mutated to P' . The logic expression that determines whether the differences between P and P' exist can be described as (2). The MP and EP mutation operators have the same logic expression as (2).

$$P \oplus P' = (P \wedge \neg P') \vee (\neg P \wedge P') \quad (2)$$

The distinguishability criterion for an IAL mutant requires that the number of iterations of the mutated loop is different in the mutant. Let us denote the condition of the original loop as L which is mutated to L' . The logic expression that determines whether the differences between L and L' exist in the i^{th} iteration can be described as (3). L_i denotes the loop condition of the original model in the i^{th} iteration. L_i' denotes the loop condition of the mutant in the i^{th} iteration. The logic expression of the execution path in the loop block during the i^{th} iteration is denoted as $block_i$. When (3) is satisfied, differences will be created in the outputs by the traversal of $block_i$ in either the original model or the mutant. The mutation operators ML and EL have the same logic expression as (3).

$$(L_1 \wedge L_1' \wedge block_1) \wedge \dots \wedge (L_{i-1} \wedge L_{i-1}' \wedge block_{i-1}) \wedge (L_i \oplus L_i') \quad (3)$$

7.4 Logic expression: Sufficiency

To kill a mutant strongly, the results of the execution of the mutant and the original SRS model need to be different. Once the necessity, reachability, and distinguishability criteria are satisfied for a mutant, the mutated component is traversed, and a difference is created by executing the mutated component. The sufficiency requires that the difference created propagates to the results of execution.

For an MF (Missing Function) mutant, a difference is created in the string that is generated by traversing the mutated function. Since the string is directly outputted to the results of execution, no extra logic expression is required to satisfy the sufficiency criterion to kill an MF mutant. The mutants from the following mutation operators do not need an extra logic expression for the sufficiency criterion for the same reason. The mutation operators are MF, EF, IAFN, FIF, MV, EV, IAVN, VIT, VIR, MIF, EIF, and IAFC.

The mutants from the other mutation operators need to satisfy the sufficiency criterion. The method to satisfy the sufficiency is: 1) generate a set of system inputs, 2) Execute the system inputs on the original SRS model and the mutant, 3) If the mutant is not killed, go to step1 and repeat the process, 4) Stop the process if the mutant is killed.

The algorithm that combines the method to satisfy the sufficiency criterion with the logic expressions of the necessity, reachability, and distinguishability criterion is given in the following section.

7.5 Algorithm to generate scenarios

In the previous sections, the logic expressions that satisfy the necessity, reachability, distinguishability, and sufficiency criteria are discussed. Table 5 lists whether a specific logic expression is required to satisfy each criterion in order to kill a mutant generated by each mutation operator. The mutation operators are divided into three groups based on their requirements of a specific logic expression for each criterion. The first column of Table 5 shows the groups of the mutation operators. The second column and third column give the mutation operators and the defect type corresponding to each mutation operator. The last four columns exhibit whether a specific logic

Table 5 Requirements of a specific logic expression for each criterion

Group	Mutation Operator	Defect name	Nec	Rea	Dis	Suf
Group 1	MF	Missing (definition of) function	Y	N	N	N
	EF	Extra (definition of) function	Y	N	N	N
	IAFN	Incorrect/ambiguous function name	Y	N	N	N
	FIF	Function with Incorrect functionality	Y	N	N	N
	MV	Missing variable	Y	N	N	N
	EV	Extra variable	Y	N	N	N
	IAVN	Incorrect/ambiguous variable name	Y	N	N	N
	VIT	Variable with incorrect type	Y	N	N	N
	VIR	Variable with incorrect range	Y	N	N	N
Group 2	MIF	Missing instance of function	Y	Y	N	N
	EIF	Extra instance of function	Y	Y	N	N
	IAFC	Incorrect/ambiguous function call.	Y	Y	N	N
Group 3	MP	Missing predicate	Y	Y	Y	Y
	EP	Extra predicate	Y	Y	Y	Y
	IAP	Incorrect/ambiguous predicate	Y	Y	Y	Y
	ML	Missing Loop	Y	Y	Y	Y
	EL	Extra Loop	Y	Y	Y	Y
	IAL	Incorrect/ambiguous loop	Y	Y	Y	Y
	MA	Missing assignment	Y	Y	Y	Y
	EA	Extra assignment	Y	Y	Y	Y
	IAA	Incorrect/ambiguous assignment	Y	Y	Y	Y

expression is required to satisfy the necessity, reachability, distinguishability, and sufficiency criterion where Y means a specific logic expression is needed and N means not needed.

Killing a mutant from the mutation operators in group 1 only requires a specific logic expression to satisfy the necessity criterion. The other criteria will be satisfied automatically. The logic expressions that satisfy both the necessity and reachability criterion are required to kill a mutant from group 2. The logic expressions of all criteria are necessary to kill a mutant from group 3.

Given a mutant, the algorithm that identifies a scenario that kills the mutant is introduced below.

The function “*GenScenario()*” in Fig. 13 takes the original model of the SRS and the mutated component of the mutant as inputs. It generates a scenario that kills the mutant. The scenario is generated based on the mutation operator that generates the mutant.

The variable “*nec_le*” is the logic expression that satisfies the necessity criterion and is obtained using the function “*model.inputRange()*” at line 1. If the mutant is from a mutation operator in group 1, a scenario that kills the mutant can be generated by simply solving the logic expression *nec_le*. The function “*solve()*” solves a logic expression and generates a scenario that fulfills the logic expression. If the function “*solve()*” returns an empty list, the logic expression has no solution.

If the mutant is from group 2, the logic expressions of all execution paths are generated at line 5 using the function “*GenAllExePathLE()*” and assigned to a set variable “*LE_set*”. The logic expressions that satisfy the reliability criterion are generated using the function *RechabilityLE()* at line 6 and assigned to a set variable *ReachabilityLE_set*. From line 7 to 10, a scenario is generated by solving the logic expressions combining the necessity and reachability criteria.

GenScenario ()	
Inputs:	
model: model of original system	
mut_comp: the mutated component	
Outputs:	
scenario: the scenario in which the mutant can be killed	
1	nec_le ← model.inputRange()
2	if mut_comp is from a group 1 mutation operator:
3	scenario ← solve(nec_le)
4	if mut_comp is from a group 2 mutation operator:
5	LE_set ← GenAllExePathLE (model, mut_comp)
6	ReachabilityLE_set ← ReachabilityLE (LE_set, mut_comp)
7	for each rel_le in ReachabilityLE_set:
9	scenario ← solve(nec_le + rel_le)
10	if the scenario is found
11	break
12	if mut_comp is from a group 3 mutation operator:
13	mut_model ← genMutant(model, mut_comp)
14	LE_set ← GenAllExePathLE (model, mut_comp)
15	SeperatePathLE ← SeperatePathLE (LE_set, mut_comp)
16	for each row in SeperatePathLE:
17	PrecedingLE, LocationLE, PosteriorLE ← obtainSubpath(row)
18	dist = model.distinguishability(mut_comp, PrecedingLE, LocationLE)
19	poss_scenario ← solve(nec_le + dist + PrecedingLE + LocationLE + PosteriorLE)
20	if a poss_scenario is found
21	compare execution results of model and mut_model in the poss_scenario
22	if the mutant is killed
23	scenario ← poss_scenario
24	break
25	return scenario

Fig. 13 Scenario Generation Algorithm

If the mutant is from a mutation operator in group 3, the mutant of the model is first generated at line 13 using a function *genMutant()* and assigned to a variable “*mut_model*”. The execution paths that traverse the mutated component are generated and separated from line 14 to line 17. The logic expression of the distinguishability criterion is generated at line 18 by a function *model.distinguishability()* and assigned to a variable “*dist*”. At line 19, the logic expression of an execution path that traverses the mutated component is combined with the logic expressions of the necessity and distinguishability criteria to solve for a possible scenario. The solution is assigned to a variable “*poss_scenario*”. The possible scenario obtained satisfies the necessity, distinguishability, and reachability criteria. At line 21, the sufficiency of the possible scenario is examined. If the mutant can be strongly killed, the possible scenario is the scenario we are looking for.

At line 25, the scenario identified is returned. It should be noted that the mutant is an equivalent mutant if no scenario is generated. This is because the algorithm searches through the input space of the system to identify the scenarios that kill the mutant. If no scenario is generated, the mutant executes in the same manner as the original model in all scenarios. In this case, the mutant is equivalent to the original model. An equivalent mutant does not need to be killed.

8 An automation tool: RITSM

An automation tool, RITSM (Requirements Inspection Tool based on Scenarios Generated by Model Mutation), was developed to help inspectors apply the RITSM method. The RITSM

tool consists of seven modules which are the SRS Input Interface, Compiler, Mutants Generator, Logic Expression Generator, Solver, Executor, and Results Processor as shown in Fig. 15. Each module is briefly introduced as follows.

The SRS input interface collects the information contained in the SRS from the user and outputs the information into a file. In the study of Li. et al. (2016), an Automated Reliability Prediction System (ARPS) tool was developed for reliability assessment of safety-critical software. ARPS allows the user to enter the information from the SRS document. Figure 14 displays the main panel of the ARPS tool. The SRS is displayed on the left side of the screen. The functions defined in the SRS can be entered into the ARPS tool using the Add Function/ Remove Function buttons in the middle portion of the panel. The detailed information for each function including inputs, outputs, function logic, etc. is collected on the right side of the panel. The logic between sub-functions and variables is specified using the “EFSM language” which shares some features with the popular procedural languages such as C. The ARPS tool is reused as the user interface of our RITSM tool. The information collected is used to construct the HLEFSM model of the SRS. The following modules are developed in this paper.

The Compiler obtains the information collected through the user interface and performs lexical analysis, syntax analysis, and semantic analysis to construct the HLEFSM model. The following defects will be detected in the semantic analysis: 1) duplicated definitions of a function, 2) the definition of a function that is never used in the system logic, 3) duplicated definitions of a variable, 4) the definition of a variable that is never used in the system logic, 5) a function instance that is never defined, and 6) a variable instance that is never defined. Once a defect is encountered in the compilation process, it will be reported in the results.

The Mutants Generator generates the mutants of the HLEFSM model following the mutation operators defined in section 6.

For each mutant, a logic expression that describes the execution path killing the mutant is generated in the Logic Expression Generator. The Logic Expression Generator interacts with

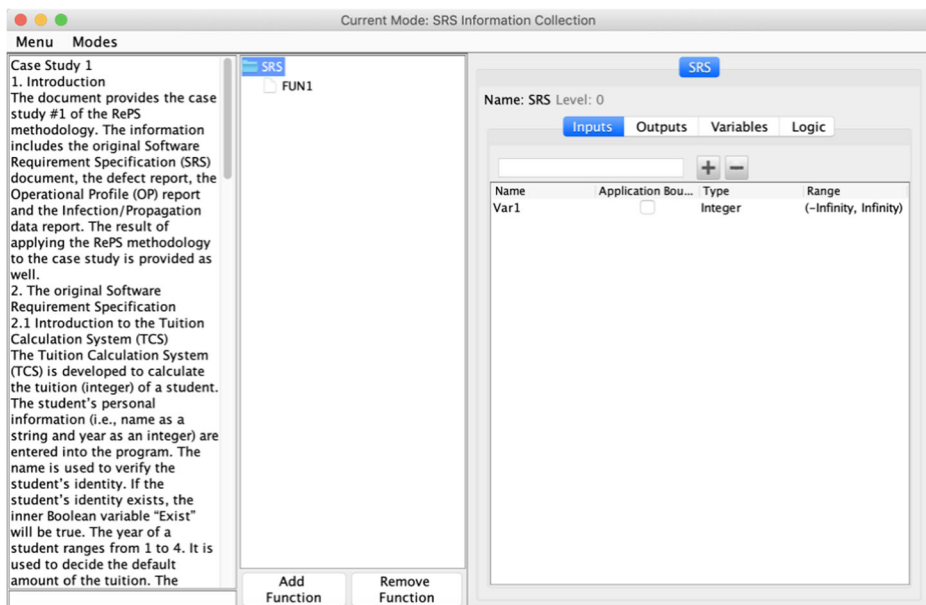


Fig. 14 The main panel of the ARPS tool for HLEFSM construction

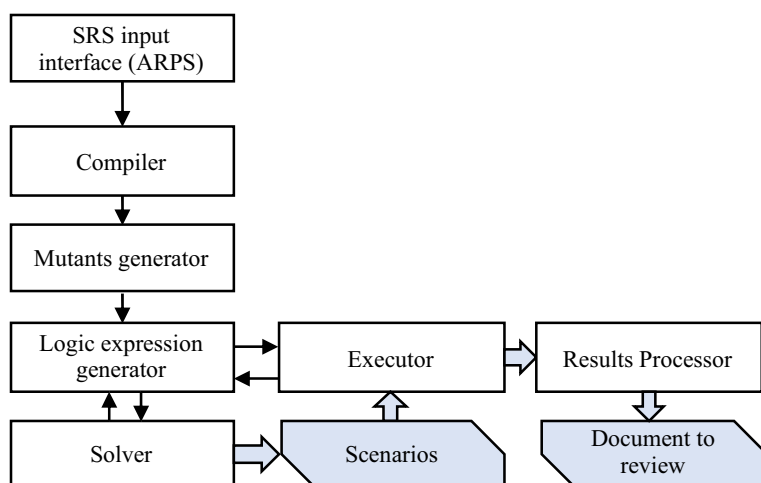


Fig. 15 RITSM Structure

the Solver and Executor. The logic expression is generated using the algorithm introduced in section 7.

The Solver solves a logic expression and generates the scenario for which the logic expression is satisfied. A logic expression is a combination of predicate conditions and assignments. Solving predicate conditions in a logic expression is a decision problem that can be solved using propositional logic and first-order logic. In this paper, these conditions are treated as Satisfiability Modulo Theories (SMT) problems. A free SMT Solver from Microsoft Research, Z3 (De Moura, Bjørner, and Bjørner 2008), is used to solve the logic expressions.

The Executor executes the HLEFSM model in a given scenario and generates the results of execution as introduced in Section 5.

The results of execution are processed by the Results Processor. A document is generated which lists the results of executing the system under each scenario.

With the help of the RITSM tool, an inspector can input the information contained in the SRS to construct the HLEFSM model. The mutants of the model and the scenarios under which the mutants can be killed are generated automatically. The RITSM tool then executes the HLEFSM model for the scenarios generated. The results of execution are outputted in a document. The inspector can analyze the behaviors and outputs of the system for each scenario and detect defects in the requirements.

9 Experimental validation of the RIMSM's performance and usability

The RIMSM method is developed to improve the performance of SRS inspection. Thus an experiment is conducted to validate the performance and usability of the RIMSM method. In the section, the RIMSM method is compared with the CBR technique which is commonly used in requirements inspection.

9.1 Research question

The questions to be answered in this experiment are:

- Determine whether the RIMSM method has a better performance than the CBR method for SRS inspection.
- Determine whether the RIMSM method is more usable than the CBR method for SRS inspection.

9.2 Research variables

The independent variables controlled in the experiment include:

- The inspection technique used by a subject (i.e. RIMSM framework and CBR method);
- The background knowledge and experience of a subject;
- The SRS documents to be inspected.

The inspection technique is our treatment variable. The other variables allow us to eliminate several potential threats to the experiment's internal validity. The dependent variables, i.e. the performance and usability measures, are defined in the following section.

9.3 Performance and usability measures

The following measures are investigated in this experiment.

9.3.1 Effectiveness

Effectiveness is a performance measure. The effectiveness of an SRS inspection method is quantified using Defect Coverage (DC). Defect Coverage is the percentage of defects identified in the SRS. It is defined in (4).

$$\text{Defect Coverage} = \frac{\text{Number of Defects identified by a subject}}{\text{Total number of defects in an SRS}} \quad (4)$$

The capture-recapture technique is used to estimate the total number of defects in an SRS. The model developed by Chao is used which assumes that different defects and different inspectors have different detection probabilities (Lee and Chao 1994). The defect population size can be estimated using (5) where N is the total number of defects estimated, D is the number of distinct defects identified by all inspectors, f_1 is the number of defects found by exactly one inspector, \hat{C} is an estimator of sample coverage (i.e. the quotient of the sum of detection probabilities of the detected defects and the sum of detection probabilities of all defects). \hat{C} is determined by the following variables $\{f_1, f_2, \dots, f_k, \dots, f_t, t\}$ where f_k is the number of defects found by exactly k inspectors and t is the total number of inspectors. $\hat{\gamma}$ is the coefficient of variation (i.e. the relative standard deviation) of the detection probabilities. $\hat{\gamma}$ is determined by the following variables $\{n_1, \dots, n_j, \dots, n_t, f_1, \dots, f_t, t\}$ where n_j is the number of defects found by the j^{th} inspector. More details about \hat{C} and $\hat{\gamma}$ can be found in (Lee and Chao 1994).

$$N = \frac{D}{\hat{C}} + \frac{f_1}{\hat{C}} \hat{\gamma}^2 \quad (5)$$

9.3.2 Time

Time is a performance measure. It is the total amount of time used to inspect an SRS. For the RIMSM method, it includes the time consumed in constructing the HLEFSM model and the time consumed in inspecting the results of execution. For the CBR method, it includes the time consumed in inspecting the SRS using a checklist.

9.3.3 Efficiency

The efficiency is also a performance measure that is defined as the average defect coverage of a subject per unit time. The equation for efficiency is given in (6).

$$Efficiency = \frac{Defect\ Coverage}{Time} \quad (6)$$

9.3.4 Satisfaction

This is a subjective measure of usability that describes how pleasant the technique is for use. Satisfaction is a measure on a four-point semantic differential scale: 1-Frustrating, 2-Unpleasant, 3 Likeable, and 4-Pleasant. The scale is defined below:

4-Pleasant: I would like to perform the inspection again without incentive.

3-Likeable: I may perform the inspection again if there is an incentive identical to the one offered in this study.

2-Unpleasant: I only want to perform the inspection again if there is much more incentive than the one offered in this study.

1-Frustrating: I will not perform the inspection again under any incentive offered.

9.3.5 Ease

This is a subjective measure of usability that describes the level of difficulty felt by the user in achieving a task. Ease is a measure on a four-point semantic differential scale: 1-Very Difficult, 2-Moderately Difficult, 3 Easy, and 4-Very Easy. The scale is defined below:

4-Very Easy: I don't feel fatigued after the inspection. I have the energy for another round of inspection now.

3-Easy: I feel fatigued after the inspection. I have some energy left, although my energy left is not enough to complete another round of inspection now.

2-Moderately Difficult: I feel very fatigued after the inspection. I don't have the energy to start another round of inspection now. I may be able to start another round of inspection after a break (several hours) today.

1-Very Difficult: I am exhausted after the inspection. I have no energy to do another round of inspection today.

9.3.6 Time pressure level

This is a subjective measure of usability that describes the degree to which the user feels time pressure in achieving a task. The time pressure level is a measure on a four-point semantic differential scale: 1- High time pressure, 2- Moderately time pressure, 3-Low time pressure, and 4-No time pressure. The scale is defined below:

- 4–No time pressure: After completing the inspection, I have enough time left to carefully review my results
- 3– Low time pressure: The time available for inspection is just enough for me to complete the inspection.
- 2– Moderate time pressure: I would like to have twice as much time to complete the inspection.
- 1–High time pressure: I would like to have four times or more as much time to complete the inspection.

9.4 Hypothesis

The null hypothesis and alternative hypothesis for each performance and usability measure are given below.

For effectiveness, the null hypothesis and alternative hypothesis are:

- $H_{0, DC}$: there is no difference between the average defect coverage of the RIMSM method and the CBR method.
- $H_{A, DC}$: the average defect coverage of the RIMSM method is higher than the CBR method.

For time, the null hypothesis and alternative hypothesis are:

- $H_{0, time}$: there is no difference between the average time consumed by the RIMSM method and the CBR method.
- $H_{A, time}$: the average time consumed by the RIMSM method is less than the CBR method.

For efficiency, the null hypothesis and alternative hypothesis are:

- $H_{0, effi}$: there is no difference between the average efficiency of the RIMSM method and the CBR method.
- $H_{A, effi}$: the average efficiency of the RIMSM method is higher than the CBR method.

The T-test is used to test the null hypothesis of effectiveness, time and efficiency based on the Central Limit Theorem.

For satisfaction, the null hypothesis and alternative hypothesis are:

- $H_{0, sat}$: the probability that the satisfaction of using the RIMSM method is higher than the CBR method is equal to the probability that the satisfaction of using the RIMSM method is lower than the CBR method

- $H_{A, sat}$: the probability that the satisfaction of using the RIMSM method is higher than the CBR method is greater than the probability that the satisfaction of using the RIMSM method is lower than the CBR method

For ease, the null hypothesis and alternative hypothesis are:

- $H_{0, ease}$: the probability that the ease of using the RIMSM method is higher than the CBR method is equal to the probability that the ease of using the RIMSM method is lower than the CBR method
- $H_{A, ease}$: the probability that the ease of using the RIMSM method is higher than the CBR method is greater than the probability that the ease of using the RIMSM method is lower than the CBR method

For time pressure level, the null hypothesis and alternative hypothesis are:

- $H_{0, time}$: the probability that the time pressure level of using the RIMSM method is higher than the CBR method is equal to the probability that the time pressure level of using the RIMSM method is lower than the CBR method
- $H_{A, time}$: the probability that the time pressure level of using the RIMSM method is higher than the CBR method is greater than the probability that the time pressure level of using the RIMSM method is lower than the CBR method

Since ranked data are collected for the measures Satisfaction, Ease, and Time Pressure Level, the nonparametric Mann-Whitney U Test (also called Wilcoxon rank-sum test) is used for hypothesis testing.

9.5 Experiment instrumentation

The instrumentations used in this experiment include the SRS documents, the checklist used in the CBR method, the tool RITSM used in the RIMSM method, and the results recording sheet. The RITSM tool has been introduced in section 8. The other three instrumentations are introduced below.

9.5.1 SRS documents

Twelve SRS documents were used in this experiment as shown in Table 6. The twelve SRS documents focus on different topic areas and are of different sizes.

The SRS S1-S6 are taken from the study by (Li, Mutha, and Smidts 2016) and reused in this experiment. The SRSs S7-S12 are taken from publicly available resources. The reference for each SRS is given in the last column of Table 6. The twelve SRSs are adapted to adhere to the IEEE suggested format. Each SRS consists of three sections: Introduction, Overview, and Specific Functions. The number of functions in each SRS and the number of pages of each SRS are intentionally controlled as shown in Table 6. Each SRS has a 12 point font and is single-spaced.

The defects in these SRSs are indigenous defects (i.e., defects that appear naturally and are not intentionally seeded). Therefore, we do not know the number of defects in

Table 6 SRS documents used in the experiment

SRS#	SRS topic	Function	Page	Defects	Ref.
S1	A Water Level Control System	4	3	5	(Li, Mutha, and Smidts 2016)
S2	A reaction chamber control system in a chemical plant	5	3	4	(Li, Mutha, and Smidts 2016)
S3	An Automated Car Assembly System	5	3	4	(Li, Mutha, and Smidts 2016)
S4	A fly safety system	5	3	3	(Li, Mutha, and Smidts 2016)
S5	A Valve Control System	5	3	5	(Li, Mutha, and Smidts 2016)
S6	A Vehicle Speed Monitor system	5	3	4	(Li, Mutha, and Smidts 2016)
S7	A Post-collision event control system	5	3	6	(Cisneros et al. 2018)
S8	An automobile cruise control and monitoring system	5	3	5	(Kirby 1987)
S9	A Digital-based small reactor protection system	10	6	9	(Santoso et al. 2019)
S10	An Elevator Control System	11	6	11	(Strobl and Wisspeintner 1999)
S11	An Integrated Vehicle-Based Safety Systems	21	11	55	(LeBlanc et al. 2008)
S12	An Embedded Control Software for Smart Sensor	22	12	22	(Derenthal et al. 2017)

each document. The total number of defects in each SRS were estimated using (5) as shown in the last column of Table 6.

SRSs S1-S8 are labeled as “Small size SRS”. SRSs S9 and S10 are labeled as “Medium size SRS”. SRSs S11 and S12 are labeled as “Large size SRS”. SRSs of different sizes are used to verify the scalability of the inspection method. The size of a medium-size SRS is about twice that of a small size SRS. The size of a large size SRS is about twice that of a medium size SRS.

For each SRS, a Stakeholder Requirements Specification (StRS) is also developed. An StRS describes the capabilities of the system that are needed by users and other stakeholders in a defined environment. The StSRs are used as oracles to guide the inspectors in the detection of defects.

9.5.2 Checklist

Traditional checklists usually cover both functional requirements and non-functional requirements. Since only the defects related to system functions are the focus of this paper, the checklist used in this paper is designed by removing the check points that are not within our scope and detailing the check points within scope using the defects taxonomy in Table 3. The checklist used is given in Fig. 16.

Category		Checkpoint (defect name)	Description
Check the definition of functions		Missing (definition of) function	Is there a function called not defined?
		Extra (definition of) function	Is there a function defined but never called?
		Incorrect/ambiguous function name	Is name of a function inconsistent (Definition and the function call are different)?
		Function with Incorrect functionality	Is functionality of a function incorrect ?
Check the definition of variables (Inputs, Outputs and Intermediate variables)		Missing variable	Is a variable used never defined?
		Extra variable	Is a variable defined never used?
		Incorrect/ambiguous variable name	Is the name of a variable inconsistent ?
		Variable with incorrect type	Is the type of a variable incorrect?
		Variable with incorrect range	Is the range of a variable incorrect?
Check the function logic	Check the function call	Missing function call	Is a function not performed by the system?
		Extra function call	Is an extra function performed?
		Incorrect/ambiguous function call.	Is an incorrect function performed?
	Check the assignments	Missing assignment	Is an assignment missing?
		Extra assignment	Is an assignment not necessary?
		Incorrect/ambiguous assignment	Is an assignment incorrect?
	Check the predicates	Missing predicate	Is a predicate missing?
		Extra predicate	Is a predicate not necessary?
		Incorrect/ambiguous predicate	Is a predicate incorrect?
	Check the loops	Missing Loop	Is a loop missing?
		Extra Loop	Is a loop not necessary?
		Incorrect/ambiguous loop	Is a loop incorrect?

Fig. 16 Checklist used in CBR

9.5.3 Results recording sheet

A results recording sheet is used to collect the results from the inspectors. An example of the results recording sheet appears in Fig. 17. On the top of a results recording sheet, the basic information collected from an inspector includes the Identification Number of the inspector, the method used in the inspection, the name of the SRS and the time consumed in the inspection.

Whenever a defect is discovered, an entry will be made on the defect recording sheet. The entry includes two types of information: Defect location (Function, Page and Line Numbers) and Defect description.

At the end of the results recording sheet, a questionnaire is placed to collect measures of Satisfaction, Ease, and Time Pressure Level.

ID: N00 Method used: CBR or RISM
 SRS name: XXX system
 Time start: XX:XX Time finish: XX:XX

Defect Recording Sheet

defect location				defect description	
#	fun#	Page#	Line#	checkpoint (defect name)	defect description (checkpoint, where is wrong)
0	3.2	1	14	Incorrect/ambiguous function name	Function name should be "Open Valve" This is only an example
0	3.3	1	16	Incorrect/ambiguous function name	Function name should be "Close Valve" This is only an example

Fig. 17 Results recording sheet

9.6 Research subject identification

In the requirements review phase of the software development lifecycle, the SRS is reviewed by inspectors belonging to the verification and validation (V&V) team. The inspectors, who are the potential users of the RIMSM method, usually have at least a bachelor's degree in engineering or related disciplines. More specifically, the potential user of the RIMSM framework should possess the following knowledge: 1) an understanding of requirements engineering; 2) the capability to program with procedural languages.

In this experiment, junior and senior undergraduate students that major in mechanical engineering, computer science, and engineering, electrical engineering are used as subjects. A qualified subject is required to know at least one programming language. Concepts related to requirements engineering were introduced in the training sessions of the experiment.

Eventually, 23 subjects were recruited in the experiment from the Ohio State University.

9.7 Experiment procedure

The experiment lasted 3 days. All sessions took place remotely using Zoom. The experiment included: 5 training sessions, 6 practice sessions, and 6 evaluation sessions. The schedule of the sessions is shown in Table 7. Four investigators were involved in the experiment. The role of each investigator is introduced below.

- Investigator 1 developed the RIMSM method and designed the experiment. Investigator 1 prepared all instrumentations and training materials used in the experiment. Investigator 1 also analyzed the data collected.
- Investigator 2 did not know the hypotheses of the experiment. The presentations in the training sessions were given by Investigator 2 to reduce research bias. The practice sessions and evaluation sessions were also hosted by Investigator 2. Investigator 2 collected the results recording sheet from the subjects at the end of each practice session and evaluation session. The identifiers and the methods used by the subjects were removed

Table 7 Experiment schedule

Day	Sessions	Content
Day 1	training session 1	Introduction of the basic concepts of requirements engineering
	training session 2	Training of the CBR method for requirements inspection
	practice session 1	Practice of the CBR method using a small size SRS (S1)
	training session 3	Training of the RIMSM method for requirements inspection
	practice session 2	Practice of the RIMSM method using a small size SRS (S2)
Day 2	training session 4	Review of the CBR method
	practice session 3	Practice of the CBR method using a small size SRS (S3)
	training session 5	Review of the RIMSM method
	practice session 4	Practice of the RIMSM method using a small size SRS (S4)
	practice session 5	Practice of the CBR method using a small size SRS (S5)
	practice session 6	Practice of the RIMSM method using a small size SRS (S6)
Day 3	evaluation session 1	Inspection of a small size SRS (S7)
	evaluation session 2	Inspection of a small size SRS (S8)
	evaluation session 3	Inspection of a medium size SRS (S9)
	evaluation session 4	Inspection of a medium size SRS (S10)
	evaluation session 5	Inspection of a large size SRS (S11)
	evaluation session 6	Inspection of a large size SRS (S12)

from the results recording sheets by Investigator 2. The results were analyzed by Investigator 1 and 3.

- Investigator 3 was not involved in the three-day experiment. Investigator 1 and 3 analyzed the results independently to assess the inter rater reliability of the data analysis.
- Investigator 4 supervised the whole experiment.

Each training session and practice session took about 45 min. There was a 15-min break between every session. There was no time limit for an evaluation session.

In the training sessions, the subjects learned the basic concepts of software requirements engineering and the two inspection methods.

The practice sessions aim to familiarize the subjects with the inspection methods learned during the training sessions. In each practice session, all subjects were assigned a small size SRS and the corresponding StRS. The subjects inspected the small size SRS using one inspection method as specified in Table 7. The performance of the subjects was supposed to have been stabilized after the practice sessions.

In the evaluation sessions, the 23 subjects were assigned into two groups randomly. Group 1 had 11 subjects and Group 2 had 12 subjects. The subjects did not know that they were grouped. The method used by each group in each evaluation session is listed in Table 8. In evaluation session 1&2, two small size SRSs were inspected. Two medium size SRSs were inspected in evaluation session 3&4. In the last two evaluation sessions, two large size SRSs were inspected.

The design in Table 8 allows us to combine the results from Group 1 and Group 2 for sessions using the SRS of the same size. For example, the results of Group 1 in evaluation session 1 can be combined with the results of Group 2 in evaluation session 2 as the results of inspecting a small size SRS using the RIMSM method. The combination provides more data points in hypothesis testing, reduces the bias caused by using a single SRS, and reduces the bias from a single group.

9.8 Threats to validity

9.8.1 Internal validity

1) Selection Bias

The research subjects are junior and senior engineering undergraduate students. Although senior students may be more knowledgeable than juniors, the difference should not affect the results. The reason for this is the fact that requirements engineering is new to all undergraduate students. The requirements inspection methods are not based on pre-existing knowledge

Table 8 Design of evaluation sessions

SRS size	Session	Group 1	Group 2
Small	evaluation session 1	RIMSM	CBR
	evaluation session 2	CBR	RIMSM
Medium	evaluation session 3	RIMSM	CBR
	evaluation session 4	CBR	RIMSM
Large	evaluation session 5	RIMSM	CBR
	evaluation session 6	CBR	RIMSM

selectively accessible to some of the subjects. Besides, the difference in subjects' majors should not affect the results. This is because learning our techniques requires an average level of programming and mathematics knowledge shared by all engineering students.

Also, the subjects were randomly assigned into two groups which will eliminate such threats.

2) Rivalry

All subjects were trained identically and did not know a priori that they would be grouped. Thus, there was no possible rivalry between the two groups during the study.

3) History

The duration of the entire experiment was only 3 days and the subjects had no other classes. Thus, the possibility that subjects experienced significant external events that would modify their attributes can be ignored.

4) Maturation

The subjects should have reached a sufficient level of maturation through the practice sessions. Therefore, maturation during the assessment sessions can be ignored.

5) Repeated testing

The SRSs prepared in the assessment sessions were of different sizes and topics, and developed by different companies which minimized the effect of repeated testing.

6) Hawthorne effect

During the assessment sessions, the subjects inspected the SRSs independently without disturbance by the investigators. Also, the subjects should experience the same Hawthorne effect for both methods. Its effect will be eliminated during the comparison of the methods.

7) Experimenter bias

Experimenter bias was minimized in all sessions since all sessions were handled by Investigator 2 who did not know the hypotheses and was not a developer of the RIMSM method.

The experimenter bias was also minimized for results analysis since Investigator 1 and 3 did not know the relation between the results and the method used during data analysis.

8) Observer-expectancy effect

The subjects were not told that the two methods were being compared as well as which method was preferred. The title given to the research study was "Study of software requirements inspection process" to prevent such bias during the study. The threat of the Observer-expectancy effect should be minimized.

9) Mortality

One subject in Group 1 dropped out of the evaluation sessions 5 and 6 due to a home emergency. This bias is minimized since the results from both groups were combined to compare the two inspection methods.

10) Instrumentation bias

The checklist used in the CBR method may affect the results of the CBR method. The checklist (as shown in Fig. 15) used in this paper focuses on the same types of faults as the RIMSM method so as to allow like for like comparison Figs. 16 and 17.

9.8.2 External validity

Our research subjects are junior and senior engineering undergraduates. This is because the average users of our method are from engineering fields with at least a bachelor's degree. Therefore, generalizing our findings should be valid since our subjects have a similar background and learning potential as the expected users. Although the expected users may be more knowledgeable than the subjects in their particular field of expertise, learning our methodologies just requires an average level of proficiency in mathematics, programming, and modeling. These characteristics are shared by the potential users and the student subjects. Also, it is expected that both students and industry users are equally novices with regard to the RIMSM method.

The six SRSs used in the evaluation sessions were designed in such a way that different types of safety critical systems were covered. Besides, the SRSs used are independent of the inspection methods. Different sizes were considered as well. Therefore, possible biases introduced in the evaluation sessions should be minimized and should not affect external validity.

9.9 Data analysis

The results of data analysis are introduced in this section including the stability of the performance of the subjects using both methods in the practice sessions, the inter rater reliability observed in data analysis, and the statistics associated with the performance and usability measures.

9.9.1 Stability of the performance of the subjects

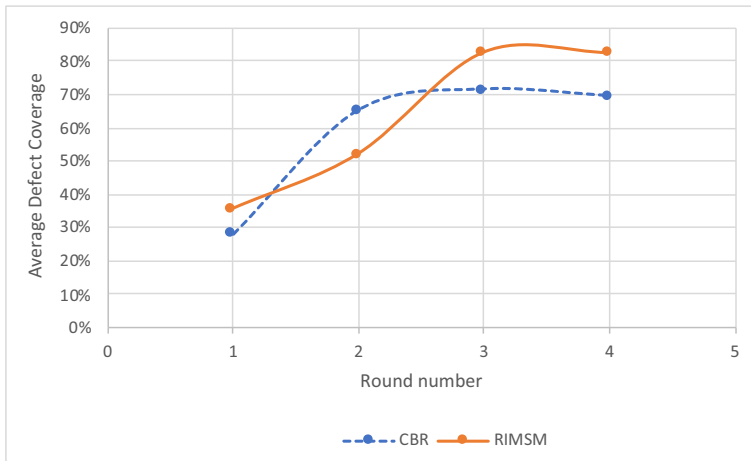
In the practice sessions and the first two evaluation sessions, eight small size SRS documents were inspected using the two methods. The average defect coverage of each inspection method is given in Table 9 and displayed in Fig. 18.

Figure 18 shows that the performance of the subjects improved in the first three rounds and became stable in the third and fourth rounds respectively. The following null and alternative hypotheses are tested to verify the stability of the performance of the subjects.

- $H_{0, RIMSM}$: there is no difference between the average effectiveness of RIMSM in the third round and the fourth round.
- $H_{A, RIMSM}$: the average effectiveness of RIMSM is different in the third round and the fourth round.

Table 9 The average defect coverage of the two methods in the first four rounds

Round#	CBR		RIMSM	
	Average defect coverage	Standard deviation	Average defect coverage	Standard deviation
1st	28.1%	7.0%	35.7%	17.4%
2nd	65.2%	27.2%	52.2%	25.2%
3rd	71.4%	21.1%	82.6%	22.0%
4th	69.4%	27.1%	82.5%	17.1%

**Fig. 18** Average defect coverage of the two methods in the first four times of usage

- $H_{0, CBR}$: there is no difference between the average effectiveness of CBR in the third round and the fourth round.
- $H_{A, CBR}$: the average effectiveness of CBR is different in the third round and the fourth round.

The T-test is used to test the null hypotheses. A 5% significance level is selected. The testing results are shown in Table 10. The p values are much greater than the significance level for both methods which means the null hypotheses can not be rejected. Since the null-hypotheses are assumed to be true for both methods, the performance of the subjects is assumed to be stable in using both methods after the third round.

9.9.2 Inter rater reliability (IRR)

Given the results recording sheet, the investigators need to determine whether the defects identified by an inspector are valid. Each entry in the results recording sheet was analyzed by Investigator 1

Table 10 Stability statistics summary

	RIMSM	CBR
Number of data points	23	23
Significance Level	0.05	0.05
p value (two-tailed)	0.98	0.78
Statistical power	5.9%	5.0%

Table 11 Inter rater reliability results

Results from	Percentage agreement	Cohen's Kappa coefficient
Evaluation session 1&2	95.7%	83.9%
Evaluation session 3&4	98.9%	97%
Evaluation session 5&6	98.2%	86.2%
Overall	97.9%	90.4%

and 3 independently. The inter rater reliability (IRR) shows the level of agreement between the investigators and the degree to which the data collected are reliable. Two methods are used to quantify the IRR: the Percentage agreement and Cohen's Kappa coefficient (L. 2012). The Percentage agreement gives the direct results of IRR and the Cohen's Kappa coefficient takes into account the possibility of the agreement occurring by chance. The results are shown in Table 11.

The second row of Table 11 shows the IRR in analyzing results from the evaluation sessions 1&2 in which two small size SRSs were inspected. The following two rows display the IRR in analyzing results from the evaluation sessions 3&4 and evaluation sessions 5&6. The overall IRR appears in the last row. The overall percentage agreement is 97.9% and the overall Cohen's Kappa coefficient is 90.4%.

Based on (McHugh 2012b), a Cohen's Kappa coefficient greater than 90% shows that the level of agreement between raters is "Almost perfect". The percentage of reliable data is between 82% and 100%.

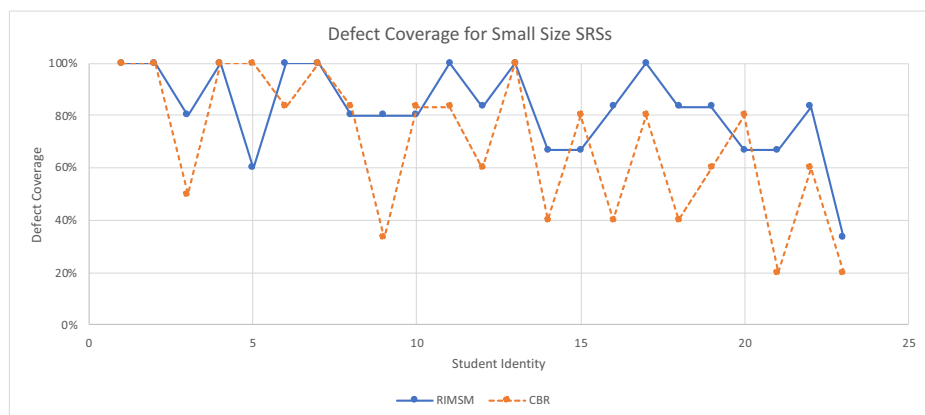
9.9.3 Effectiveness

The defect coverage achieved by each subject using each method for SRSs of the small, medium, large size is plotted in Fig. 19, Fig. 20 and Fig. 21.

A traditional t-test is used to test the null hypothesis $H_{0, DC}$ for the inspection of SRSs of each size. The statistics are summarized in Table 12. The significance level is 0.05.

The null hypothesis $H_{0, DC}$ is rejected for the inspection of SRSs of each size since the p value is smaller than the significance level 0.05 in each case as shown in row 6 of Table 12. The results show that the RISM method has a better performance in terms of effectiveness than the CBR method.

As the size of the SRS increases, the difference between the two methods becomes more obvious. For small size SRSs, the RISM increases the average defect coverage by 19%. The

**Fig. 19** Defect coverage for small size SRSs

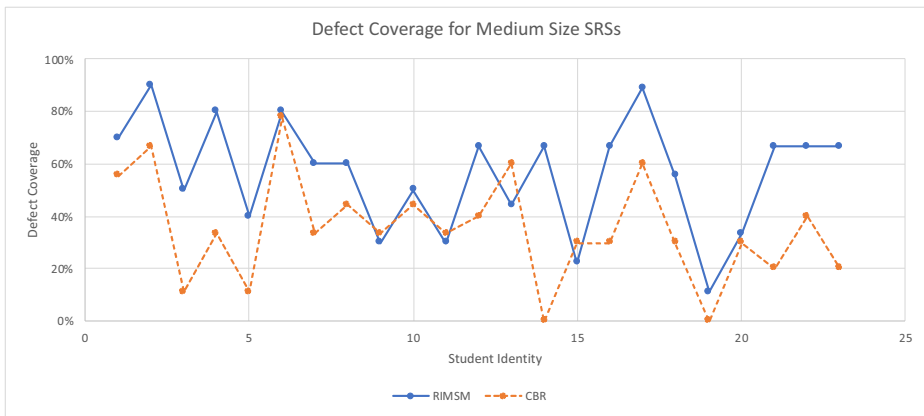


Fig. 20 Defect coverage for Medium size SRSs

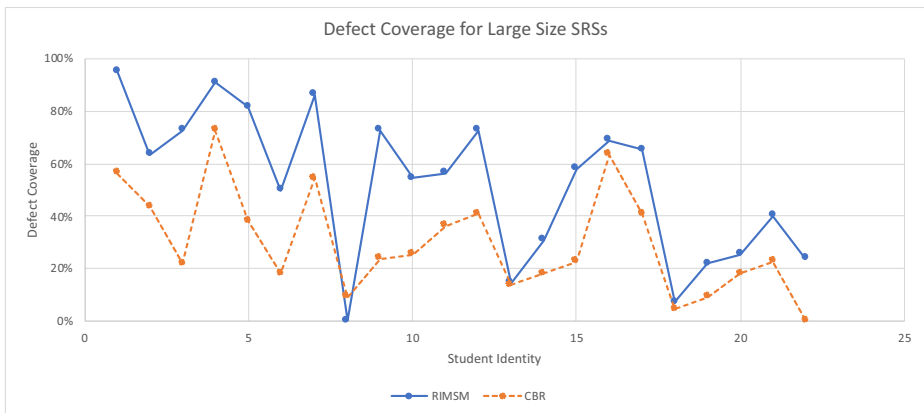


Fig. 21 Defect coverage for Large size SRSs

RIMSM increases the average defect coverage by 61% and 76% for medium size SRSs and large size SRSs respectively. Row 5 provides the 95% confidence interval of the average defect coverage. The confidence intervals are plotted in Fig. 22. The confidence intervals also suggest that the RIMSM method is more effective.

Table 12 Statistics summary for defect coverage

Row#	Size	Small		Medium		Large	
		RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
1	#data points	23	23	23	23	22	22
2	Mean	82.5%	69.4%	56.3%	35.0%	52.4%	29.8%
3	Standard Deviation	17.1%	27.1%	21.1%	20.1%	28.1%	19.7%
4	Coefficient of variation	20.7%	39.0%	37.4%	57.5%	53.5%	66.1%
5	Confidence interval (95%)	(0.751, 0.899)	(0.577, 0.811)	(0.472, 0.654)	(0.263, 0.437)	(0.402, 0.646)	(0.213, 0.383)
6	p value (one tail)	0.029		0.00048		0.0018	
7	Statistical power	62.4%		96.9%		92.6%	
8	Effect size	0.59		1.05		0.95	

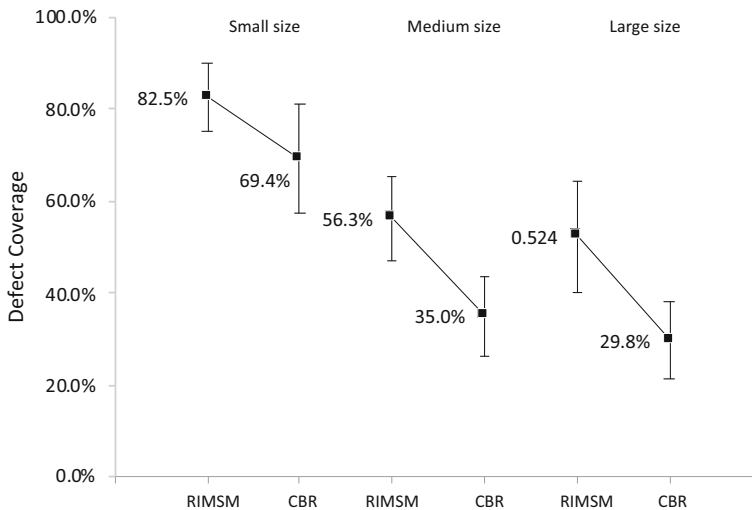


Fig. 22 Confidence interval for the average defect coverage

The high effectiveness of the RIMSM method can be attributed to the following reasons. First, an inspector only needs to analyze the behaviors and outputs of the system in one scenario each time. Identifying a defect related to system logic is easier. Second, the RITSM tool helps identify defects related to the definitions of functions and variables when compiling the HLEFSM model. The defects will be reported in the results of execution.

The standard deviations of the defect coverage in Row 3 do not exhibit a clear trend between the two methods. Row 4 gives the coefficient of variation (i.e. the relative standard deviation). It shows that the RIMSM method has a lower coefficient of variation for the SRSs of each size. In another word, the defect coverage of the RIMSM method has a lower level of dispersion around the mean. It means that the subjects have a relatively more stable performance in terms of effectiveness using the RIMSM method.

Row 7 displays the statistical power of the test. For medium and large size SRSs, the probability of a type-II error is very low.

Cohen's *d* is used to measure the effect size of the two methods for each measure in this paper. Effect size shows the magnitude of the difference between the two methods for each measure (Sullivan and Feinn 2012). An effect size below 0.2 suggests that the difference between the two methods is very small. An effect size between 0.2 and 0.5 means that the difference between the two methods is small. An effect size between 0.5 and 0.8 suggests a medium difference between the two methods. The difference between the two methods is large if the effect size is greater than 0.8 (Sawilowsky 2009).

Row 8 in Table 12 displays the effect size of the two methods in terms of effectiveness. For a small size SRS, the difference between the effectiveness of the two methods is medium. For a medium size or a large size SRS, the difference between the effectiveness of the two methods is large.

9.9.4 Time

The time consumed by each subject using each method for SRSs of each size is displayed in Fig. 23, Fig. 24, and Fig. 25.

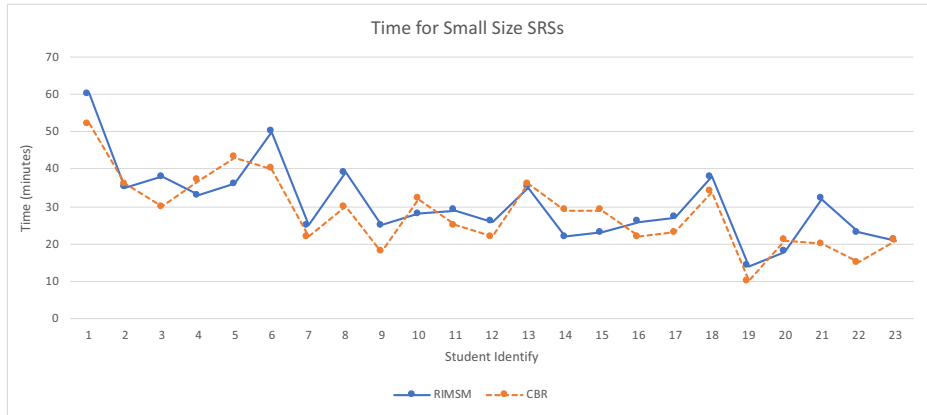


Fig. 23 Time for small size SRSs

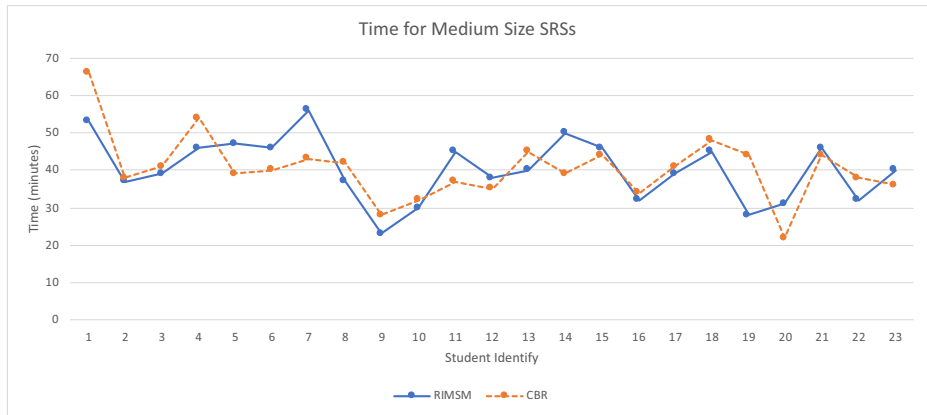


Fig. 24 Time for medium size SRSs

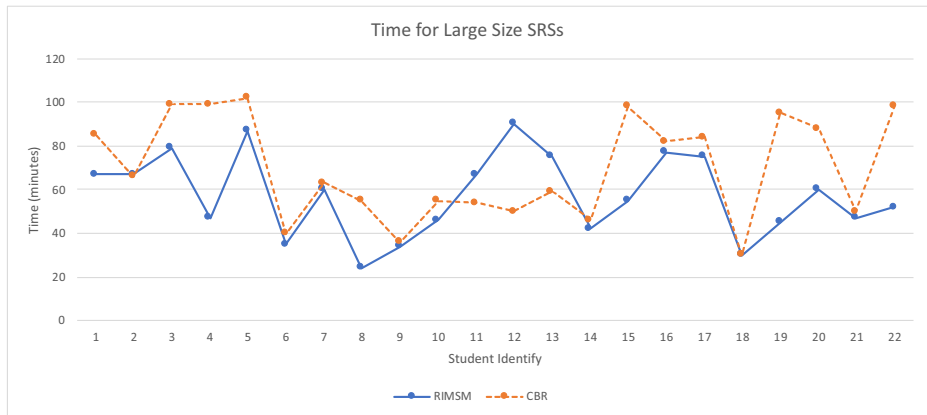


Fig. 25 Time for large size SRSs

Table 13 Statistics summary for time

Row#	Size	Small		Medium		Large	
		RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
1	#data points	23	23	23	23	22	22
2	Mean (minutes)	30.5	28.1	40.2	40.4	57.3	69.7
3	Standard Deviation	10.3	9.8	8.3	8.6	18.7	23.6
4	Coefficient of variation	33.7%	34.8%	20.7%	21.4%	32.7%	33.8%
5	Confidence Interval (95%)	(26.0, 34.5)	(23.8, 32.3)	(36.6, 43.8)	(36.7, 44.1)	(49.0, 65.6)	(59.2, 80.1)
6	p value (one tail)	0.21		0.47		0.03	
7	Statistical power	20.2%		5.9%		61.9%	
8	Effect size	0.24		0.02		0.60	

The time statistics are summarized in Table 13. Using a significance level of 0.05, the null hypothesis $H_{0, time}$ is not rejected for the inspection of the small size and medium size SRSs (See p value in Row 6). $H_{0, time}$ is rejected for the inspection of the large size SRSs.

The results tell that there is no difference between the average time consumed by both methods when inspecting a small size or medium size SRS. The RIMSM method requires less time than the CBR method when inspecting a large size SRS. The confidence intervals in Row 5 suggest the same conclusion.

This can be attributed to the following reasons. First, the RIMSM method consumes inspectors' time to enter the information of an SRS into the RITSM tool to construct the HLEFSM model before detecting defects. The average percent of the time that is consumed in constructing the HLEFSM model is 57%, 64%, and 70% of the total time consumed for inspecting the small size, medium size and large size SRSs respectively. On the other hand, the RIMSM method saves inspectors' time to detect defects once the model is constructed. As the size of the SRSs increase, detecting a defect consumes more time. Thus, the time consumed in inspecting a small size or medium size SRS using both methods is similar. The average time consumed by the RIMSM method is 17.8% less than the CBR method for inspecting the large size SRSs.

The coefficient of variation in Row 4 shows that the time consumed by the two methods has the same level of dispersion around the mean for the SRSs of each size. Row 7 gives the statistical power of the test. The power of testing the null hypothesis $H_{0, time}$ for the large size SRSs is 61.9%. Row 8 displays the effect size of the time consumed using both methods. For a small size SRS, the difference between the two methods is small. For a medium size SRS, the difference between the two methods is very small. The difference between the two methods is medium when inspecting a large size SRS.

9.9.5 Efficiency

The efficiency of each subject using each method for SRSs of each size is displayed in Fig. 26, Fig. 27, and Fig. 28.

The efficiency statistics are summarized in Table 14.

Based on the p value in row 6 of Table 14, the null hypothesis $H_{0, effc}$ is not rejected for the small size SRS. The average efficiencies for inspecting a small size SRS using both methods are similar as shown in row 2. The reason for this is that the time consumed in inspecting a

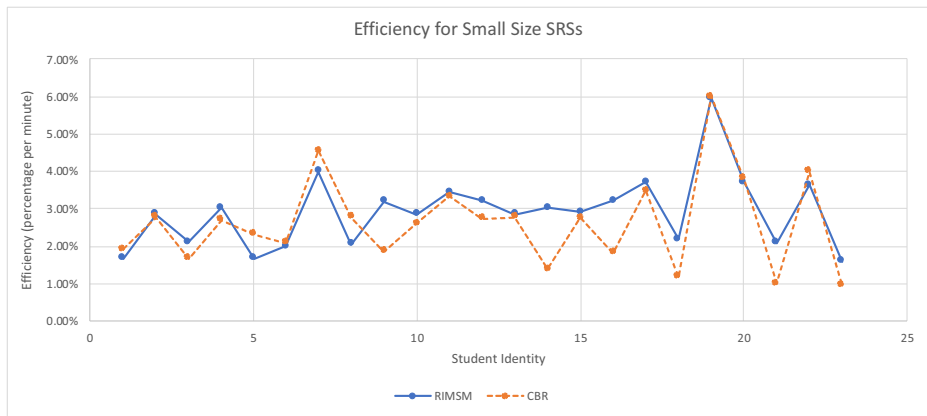


Fig. 26 Efficiency for small size SRSs

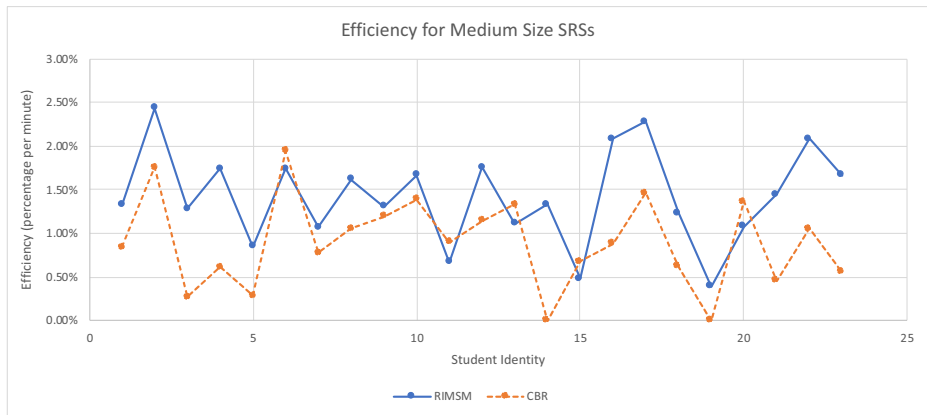


Fig. 27 Efficiency of medium size SRSs

small size SRS using the RIMSM method is slightly larger than the time necessary to inspect the SRS by the CBR method (see Table 13). Although the RIMSM method exhibits higher

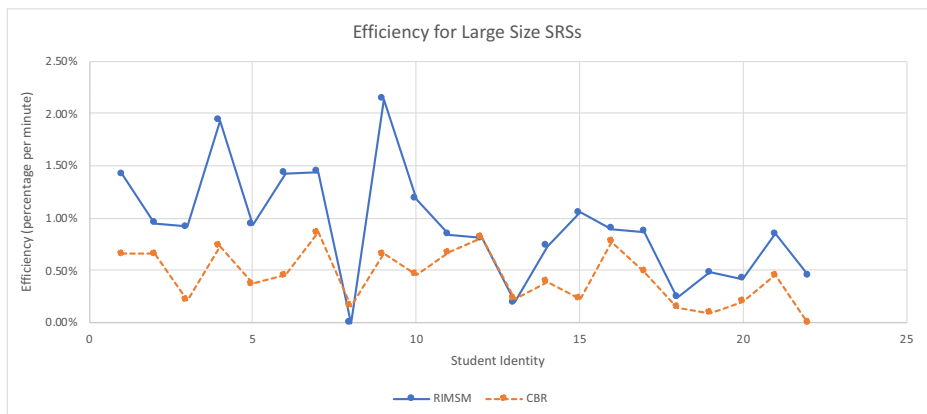


Fig. 28 Efficiency of large size SRSs

Table 14 Efficiency statistics summary

Row#	Size	Small		Medium		Large	
		RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
1	#data points	23	23	23	23	22	22
2	Mean	2.91%	2.63%	1.42%	0.89%	0.92%	0.44%
3	Standard Deviation	0.98%	1.20%	0.54%	0.52%	0.53%	0.26%
4	Coefficient of variation	33.62%	45.76%	37.87%	57.89%	57.81%	57.95%
5	Confidence Interval (95%)	(2.49%–3.33%)	(2.11%–3.15%)	(1.19%–1.64%)	(0.67%–1.12%)	(0.69%–1.15%)	(0.33%–0.55%)
6	p value (one tail)	0.20		0.00074		0.00036	
7	statistical power	21.8%		96.0%		98.8%	
8	Effect size	0.26		1.02		1.17	

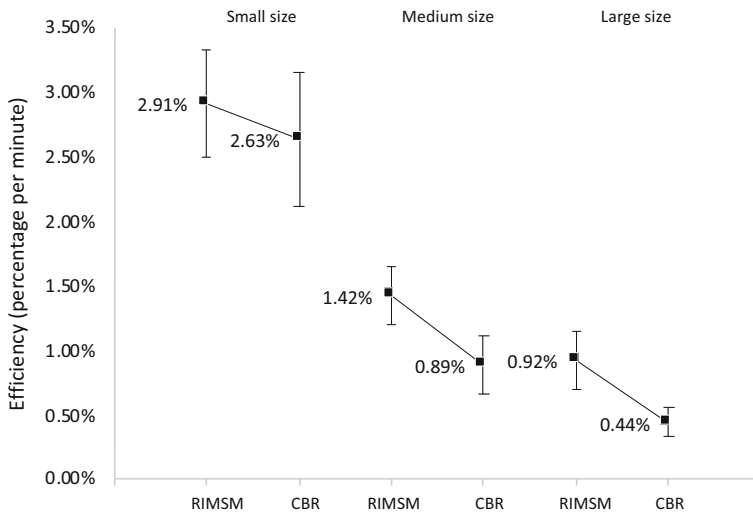


Fig. 29 Confidence interval of average defect coverage

effectiveness (see Table 12), the efficiency of the two methods is close to each other for small size SRSs Table 14.

The null hypothesis $H_{0, effc}$ is rejected for the medium size SRSs since the RIMSM exhibits higher effectiveness and requires the same amount of time for inspecting the SRS than the CBR. The null hypothesis $H_{0, effc}$ is rejected for the large size SRSs because the RIMSM exhibits higher effectiveness and requires less time for inspecting an SRS than the CBR.

The results show that for a small size SRS, there is no difference between the RIMSM method and the CBR method in terms of efficiency. However, the RIMSM method shows higher efficiency than the CBR method for the medium size and large size SRSs. The average efficiencies of both methods are shown in row 2. Using RIMSM increases the average efficiency by 59% and 107% for the medium size SRS and large size SRS respectively. Row 5 provides the 95% confidence interval of the average efficiency in each case. The confidence intervals are plotted in Fig. 29.

The standard deviations of the efficiency in Row 3 do not show a clear trend between the two methods. The coefficient of variation in Row 4 shows that the efficiency of the RIMSM method has a lower level of dispersion around the mean. It means that the subjects have a relatively more stable performance in terms of efficiency when they use the RIMSM method.

Row 7 displays the statistical power of the test. For medium and large size SRSs, the power of the test is high.

The effect size in Row 8 suggests that the difference between the efficiency of the two methods is small for inspecting a small size SRS, and large for inspecting a medium size or a large size SRS.

9.9.6 Satisfaction

Ranked data are collected for the Satisfaction measure using a questionnaire. Table 15 displays the number of subjects that select each rank value in the inspection of the SRSs of each size.

The Mann-Whitney U test is used for hypothesis testing. The results are summarized in Table 16. Given a 5% significance level, the null hypothesis $H_{0, sat}$ is not

Table 15 Satisfaction data

Rank	Small		Medium		Large	
	RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
4	6	4	6	2	3	2
3	15	12	13	13	13	6
2	2	7	4	8	6	9
1	0	0	0	0	0	5

Table 16 Satisfaction statistics summary

	Small size		Medium size		Large size	
	RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
#data points	23	23	23	23	22	22
Average Rank Value	3.17	2.87	3.09	2.74	2.86	2.23
p Value (one tail)	0.08		0.06		0.01	
Statistical Power	47.9%		56.2%		83.4%	
Effect size	0.49		0.55		0.82	

rejected for the small size and medium size SRSs, and is rejected for the large size SRSs. The results show that there is no significant difference between the two inspection methods in terms of satisfaction for small size and medium size SRSs. The experience of inspecting a large size SRS was more satisfactory when the subjects used the RIMSM method than when they used the CBR method. Table 16 also gives the power of the hypothesis testing for the SRSs of each size. The power of the Mann-Whitney U test is calculated following the procedures in (NCSS Statistical Software 2020). The power of the test is high for the large size SRSs. The effect size shows that the difference between the satisfaction of using the two methods is small for a small size SRS, medium for a medium size SRS and large for a large size SRS.

9.9.7 Ease

Table 17 displays the number of subjects that select each rank value for the Ease measure in the inspection of SRSs of each size.

Table 17 Ease data

Rank	Small		Medium		Large	
	RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
4	11	10	4	2	2	1
3	10	9	16	11	6	3
2	2	4	3	10	12	9
1	0	0	0	0	2	9

Table 18 Ease statistics summary

	Small size		Medium size		Large size	
	RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
#data points	23	23	23	23	22	22
Average Rank Value	3.39	3.26	3.04	2.65	2.36	1.82
p Value (one tail)	0.31		0.03		0.02	
Statistical Power	15.2%		69.5%		69.9%	
Effect size	0.19		0.66		0.68	

The results of hypothesis testing for ease are summarized in.

Table 18. Given a 5% significance level, the null hypothesis $H_{0, ease}$ is not rejected for the small size and is rejected for the medium size and large size SRSs. The results show that the subjects did not feel a difference between the methods in terms of ease in achieving the inspection task for the small size SRSs. The subjects felt a lower level of difficulty using the RIMSM method in achieving the inspection task for the medium size and large size SRSs.

The power of the test is medium for the medium and large size SRSs.

The effect size shows that the difference between the ease of using the two methods is very small for inspecting a small size SRS, and medium for inspecting a medium size SRS or a large size SRS.

9.9.8 Time pressure level

Table 19 displays the number of subjects that select each rank value for the time pressure level measure in the inspection of SRSs of each size.

The results of hypothesis testing for the time pressure level are summarized in Table 20. Given a 5% significance level, the null hypothesis $H_{0, time}$ is not rejected for

Table 19 Time pressure level data

Rank	Small		Medium		Large	
	RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
4	13	9	10	7	6	1
3	8	13	10	9	6	6
2	2	1	2	6	8	6
1	0	0	1	1	2	9

Table 20 Time pressure level statistics summary

	Small size		Medium size		Large size	
	RIMSM	CBR	RIMSM	CBR	RIMSM	CBR
#data points	23	23	23	23	22	22
Average Rank Value	3.48	3.35	3.26	2.96	2.73	1.95
p Value (one tail)	0.21		0.12		0.01	
Statistical Power	17.3%		32.9%		83.2%	
Effect size	0.21		0.37		0.82	

the small size and medium size SRSs, and is rejected for the large size SRSs. The results show that there is no significant difference between the two inspection methods in terms of time pressure for the small size and medium size SRSs. The subjects experienced a lower time pressure when using the RIMSM method in comparison to the time pressure they experienced when using CBR for the large size SRSs.

Based on the effect size, the difference between the time pressure level when using the two methods is small for a small size SRS and a medium size SRS. The difference is large for a large size SRS.

Sections 9.9.5–9.9.7 compare the usability of the RIMSM method to the CBR method. The following approaches can be considered to further improve the usability of the RIMSM method. First, the RIMSM method requires the users to construct the SRS model using the RITSM tool manually. A more user-friendly interface can be developed to help in the construction of the SRS model. Natural language processing techniques can be applied to help or automate the process of model construction. Second, the RITSM tool outputs the results of execution in a document which is used to detect defects. The RITSM tool can be updated to display the results of execution in a way that eases the detection of defects and of their locations. For example, the function logic executed can be displayed using a diagram to help the user determine its correctness.

9.9.9 Scalability

The results of the data analysis are summarized in this section. Table 21 displays whether the null hypothesis related to a measure is rejected for the inspection of the SRSs of each size. In the inspection of a small size SRS, only the null hypothesis of the effectiveness is rejected. In the inspection of a medium size SRS, the null hypotheses of the efficiency and ease measure are rejected in addition. In the inspection of a large size SRS, all null hypotheses are rejected. As the size of an SRS increases, the RIMSM method displays a large number of properties.

Table 22 provides the improvement in the mean of each measure which is observed when using the RIMSM method. It shows that the improvement brought by the RIMSM method increases as the size of the SRSs increases for all measures. The most significant improvement is observed for effectiveness and efficiency for the medium size and large size SRSs.

Table 21 Rejection of the null hypotheses for SRSs of each size

Measure	Small	Medium	Large
Effectiveness	rejected	rejected	rejected
Time	/	/	rejected
Efficiency	/	rejected	rejected
Satisfaction	/	/	rejected
Ease	/	rejected	rejected
Time pressure level	/	/	rejected

Table 22 Improvement in the mean of each measure using the RISM method

Measure	Small	Medium	Large
Effectiveness	18.8%	61.0%	76.3%
Time	/	/	17.8%
Efficiency	/	58.6%	106.8%
Satisfaction	/	/	28.6%
Ease	/	14.8%	30.0%
Time pressure level	/	/	39.5%

10 Conclusion

In this paper, a requirements inspection method based on scenarios generated by model mutation is proposed to detect defects in the functional requirements of a safety-critical system. The RISM method models software requirements using a High Level Extended Finite State Machine. A method that executes the HLEFSM model is defined. The method uncovers the behaviors and generates the outputs of the system for a given scenario. To identify an adequate set of scenarios in which the model shall be executed, the concept of traditional mutation is extended to the requirements phase. Twenty-one mutation operators are designed based on a taxonomy of defects defined for the requirements phase. Mutants of the HLEFSM model are generated using these operators. Further, an algorithm is developed to identify scenarios that can kill the mutants. The set of scenarios is considered to be adequate for detecting defects in the model when all mutants generated are killed.

An automation tool, RITSM, is developed to help inspectors apply the RISM method. The RITSM tool takes inputs from the inspectors to construct the HLEFSM model of an SRS. The mutants of the model and the scenarios by which the model mutants can be killed are generated automatically. The RITSM tool then executes the HLEFSM model in the generated scenarios and lists the behaviors and the outputs of the system for each scenario. By inspecting the results of the RITSM tool, the defects in the HLEFSM model can be detected and then their location identified in the SRS.

The performance and usability of the RISM method are verified in an experiment. The RISM method displays a large number of properties. The most significant properties of the RISM method are that it improves the effectiveness of inspecting the SRSs of different sizes, and it improves the efficiency of inspecting the medium size and large size SRSs. The results of the experiment allow the conclusion that the RISM method has improved the performance and usability of SRS inspection.

Future research includes the following aspects. First, the RISM method can be further extended to the design phase to help verify the software design document. Second, the scenarios generated in the requirements phase can be further studied and used in the testing phase to help verify the correctness of the source code. Third, the experimental validation of the RISM method can be conducted with more subjects from industry. Fourth, the RISM method can be compared to other inspection methods such as the Ad Hoc Reading and the Perspective Based Reading for further validation. Fifth, the performance of the RISM method in an inspection group with multiple inspectors can be studied.

Acknowledgments This material is based upon work supported by the U.S. Department of Energy, Office of Nuclear Energy, Nuclear Energy Enabling Technologies (NEET) program under the Award Number DE-

NE0008434. This research is performed using funding received from the DOE Office of Nuclear Energy's Nuclear Energy University Program under the Award Number DE-NE308896. This research is also funded by the Department of Mechanical and Aerospace Engineering, Ohio State University.

References

- Agrawal H, DeMillo RA, Hathaway B, Hsu W, Wynne H, Krauser EW, Martin RJ, Mathur AP, Spafford E. (1989) "Design of Mutant Operators for the C programming language." Technical report SERC-TR-41-P, Software Engineering Research Center, Purdue University
- Ali SW, Ahmed QA, Shafi I. (2018). "Process to Enhance the Quality of Software Requirement Specification Document." *2018 International Conference on Engineering and Emerging Technologies*
- Alshazly AA, Elfatary AM, Abougabal MS (2014) Detecting defects in software requirements specification. *Alexandria Engineering Journal* 53(3):513–527
- Arndt SA, Alvarado R, Dittman B, Mott K, Wood R (2017) "NRC Technical Basis For Evaluation Of Its Position On Protection Against Common Cause Failure In Digital Systems Used In Nuclear Power Plants." In *Proceedings of 2017 NPIC-HMIT*
- Budd TA, Gopal AS (1985) Program testing by specification mutation. *Comput Lang* 10(1):63–73. [https://doi.org/10.1016/0096-0551\(85\)90011-6](https://doi.org/10.1016/0096-0551(85)90011-6)
- Budd TA, DeMillo RA, Lipton RJ, Sayward FG (1978) The Design of a Prototype Mutation System for program testing. *Proceedings of the AFIPS National Computer Conference* 74:623–627
- Cisneros A, Rafael J, Garcia GDLR, and Fernandez-Y-Fernandez CA (2018) "Software Requirement Specification for the Automotive Sector: The Case of a Post-Collision Event Control System." *Proceedings of 5th International Conference in Software Engineering Research and Innovation, IEEE*
- Dalpiaz F, van der Schalk I, Brinkkemper S, Aydemir FB, Lucassen G (2019) Detecting terminological ambiguity in user stories: tool and experimentation. *Inf Softw Technol* 110:3–16. <https://doi.org/10.1016/j.infsof.2018.12.007>
- Demillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection : help for the practicing programmer. *Computer* 11(4):33–41
- Derenthal FE, Elks CR, Bakker T, and Fotouhi M (2017) "Virtualized Hardware Environments for Supporting Digital I&C Verification." *Proceedings of 2017 NPIC-HMIT*, 1658–70
- Fabbri SCPF, Maldonado JC, Sugeta T, Masiero PC (1999) Mutation Testing Applied to Validate Specifications Based on Statecharts. *Proceedings of 10th International Symposium on Software Reliability Engineering, IEEE*:210–219. <https://doi.org/10.1109/ISSRE.1999.809326>
- Hamlet RG (1977) Testing programs with the aid of a compiler. *IEEE Trans Softw Eng* 4:279–290
- He L, Carver J (2006) PBR vs. Checklist: A Replication in the N-Fold Inspection Context. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*:95–104. <https://doi.org/10.1145/1159733.1159750>
- Hierons RM, Merayo MG (2007) "Mutation testing from probabilistic finite state machines." *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques*, 141–50. <https://doi.org/10.1109/TAIC.PART.2007.20>
- Hierons R, Merayo M (2009) Mutation testing from probabilistic and stochastic finite state machines. *J Syst Softw* 82(11):1804–1818. <https://doi.org/10.1016/j.jss.2009.06.030>
- IEEE (2018) Systems and software engineering-life cycle processes: requirements engineering. ISO/IEC/IEEE 29148:2018
- Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. *IEEE Trans Softw Eng* 37(5):649–678
- Kim SW, Clark J, McDermid J (1999) "The rigorous generation of Java mutation operators using HAZOP." *Proceedings of the 12th international conference on software and systems engineering and their applications ICSSEA99*, 1–20
- King KN, Jefferson Offutt A (1991) A Fortran language system for mutation-based software testing. *Software: Practice and Experience* 21(7):685–718. <https://doi.org/10.1002/spe.4380210704>
- Kirby J (1987) "Software requirements for an automobile cruise control and monitoring system." *NRL-SCR*
- Laitenberger O (2002) A survey of software inspection technologies. *Handbook Softw Eng Knowl Eng* 2:517–555
- Lanubile F, Visaggio G (2000) "Evaluating Defect Detection Techniques for Software Requirements Inspections." *International Software Engineering Research Network Report No. 00–08*
- LeBlanc D, Bezziina D, Tiernan T, Gabel M, Pomerleau D (2008) "Functional requirements for integrated vehicle-based safety system (IVBSS)." *The University of Michigan Transportation Research Institute, Visteon Corporation and Cognex Corporation*

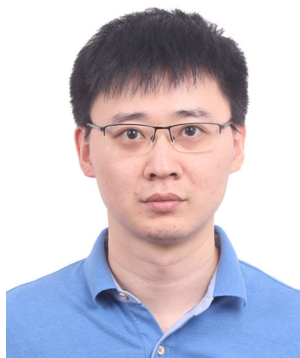
- Lee S-M, Chao A (1994) Estimating population size via sample coverage for closed capture-recapture models. *Biometrics* 50(1):88–97
- Li B, Smidts C, Huang F, Quinn T (2018) “A Quantification Framework for Software Safety in the Requirements Phase: Application to Nuclear Power Plants.” In *Proceedings of 2018 Annual Reliability and Maintainability Symposium (RAMS)*
- Li B, Smidts CS (2017) “Extension of Mutation Testing for the Requirements and Design Faults.” In *Proceedings of 2017 NPIC-HMIT*
- Li X, and Gupta J (2013) “ARPS: An Automated Reliability Prediction System Tool for Safety Critical Software.” *PSA*, 22–27
- Li X, Mutha C, Smidts CS (2016) An automated software reliability prediction system for safety critical software. *Empir Softw Eng* 21(6):2413–2455. <https://doi.org/10.1007/s10664-015-9412-6>
- Lipton RJ (1971) “Fault diagnosis of computer programs.” Student Report, Carnegie Mellon University
- Lorber F, Larsen KG, Nielsen B (2018) “Model-based mutation testing of real-time systems via model checking.” *Proceedings of 2018 IEEE 11th international conference on software testing, verification and validation workshops (ICSTW)*, 59–68. <https://doi.org/10.1109/ICSTW.2018.00029>
- Ma YS, Offutt J, Kwon YR (2005) MuJava: an automated class mutation system. *Software Testing Verification and Reliability* 15(2):97–133. <https://doi.org/10.1002/stvr.308>
- McHugh ML (2012b) Interrater reliability: the kappa statistic. *Biochemia Medica*: Biochemia Medica 22(3)
- De Moura L, Björner N, Björner N (2008) Z3: an efficient SMT solver. In: *International conference on tools and algorithms for the construction and analysis of systems*, pp 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
- NCSS Statistical Software (2020) “Mann-Whitney U or Wilcoxon Rank-Sum Tests.” *NCSS, LLC. Kaysville, Utah, USA*, 503–15
- Okun V. 2004. “Specification mutation for test generation and analysis.” Dissertation, University of Maryland Baltimore County
- Pinto Ferraz Fabbri SC, Delamaro ME, Maldonado JC, Masiero PC (1994) Mutation Analysis Testing for Finite State Machines. *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*: 220–229. <https://doi.org/10.1109/ISSRE.1994.341378>
- Porter AA, Votta LG, Basili VR (2002) Comparing detection methods for software requirements inspections: a replicated experiment. *IEEE Trans Softw Eng* 21(6):563–575. <https://doi.org/10.1109/32.391380>
- Santoso S, Sudamo R, Maerani JS, Cahyono A (2019) Software requirement analysis for digital based reactor protection system of RDE design. *J Phys Conf Ser* 1198(2). <https://doi.org/10.1088/1742-6596/1198/2/022046>
- Sawilowsky SS (2009) New Effect Size Rules of Thumb. *Journal of Modern Applied Statistical Methods* 8(2): 597–599. <https://doi.org/10.22237/jmasm/1257035100>
- Staron M, Kuzniarz L, Thum C (2005) An empirical assessment of using stereotypes to improve Reading techniques in software inspections. *Proceedings: international conference on software engineering*:63–69. <https://doi.org/10.1145/1083292.1083308>
- Strobl F, Wisspeintner A. 1999. “Specification of an elevator control system.” Technische Universitat Munchen
- Sugeta T, Maldonado JC, Eric Wong W (2004) Mutation testing applied to validate SDL specifications. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2978:193–208. https://doi.org/10.1007/978-3-540-24704-3_13
- Sullivan GM, Feinn R (2012) Using effect size—or why the P value is not enough. *Journal of Graduate Medical Education* 4(3):279–282. <https://doi.org/10.4300/jgme-d-12-00156.1>
- Yang XM. 2007. “Towards a self-evolving software defect detection process.” Dissertation, University of Saskatchewan
- Zhang H, Yue T, Ali S, Liu C (2016) Towards mutation analysis for use cases. *Proceedings - 19th ACM/IEEE international conference on model driven engineering languages and systems, MODELS 2016*:363–373. <https://doi.org/10.1145/2976767.2976784>
- Zhang Z, Li K, Yuan L, Guanhua Y (2019) Mutation Model-Based Test Case Generation of Chinese Train Control System with Automatic Train Operation Function. *Proceedings of 2018 International Conference on Intelligent Rail Transportation (ICIRT)*. <https://doi.org/10.1109/ICIRT.2018.8641582>



Boyuan Li , Ph.D. Postdoctoral Researcher, The Ohio State University. Boyuan Li received his Ph.D. from the Nuclear Engineering Program at the Ohio State University (OSU). Before joining to OSU, he obtained his BS and MS degree in nuclear engineering program at the University of Technology and Science of China. His research interests include software requirements qualification, software dependability assessment of safety-critical systems, common cause failure analysis, fault diagnosis and sensor deployment.



Xiaoxu Diao , Ph.D. Postdoctoral Researcher, The Ohio State University. Xiaoxu Diao is a post-doctoral researcher working in the Reliability and Risk Laboratory in the Department of Mechanical and Aerospace Engineering, The Ohio State University. He received the M.Tech and Ph.D. degrees in Software Reliability Engineering from the School of Reliability and System Engineering at Beihang University, Beijing, China, in 2006 and 2015, respectively. He has participated in several research and projects regarding software reliability and testing. His research interests are real-time embedded systems, software testing methods, and safety-critical systems.



Wei Gao . Ph.D. student, The Ohio State University. Wei Gao received his MS at Shanghai Nuclear Engineering Research and Design Institute. He has nine years of experience in probabilistic risk analysis in the nuclear industry. His research interests include risk-informed application, reliability data analysis, machine learning, risk management and software development.



Carol Smidts , Professor, IEEE Fellow. Professor, The Ohio State University. Carol S. Smidts is a professor in the Department of Mechanical and Aerospace Engineering at The Ohio State University, since 2008. Prior to this appointment, she was an associate professor in the Reliability Engineering Program at the University of Maryland at College Park, and the director of the Software Reliability Engineering curriculum. She is an expert in software reliability and probabilistic risk assessment. Her research interests include software reliability modeling, software test automation, probabilistic dynamics for complex systems, and human reliability. She was the conference program committee co-chair of the International Symposium on Software Reliability Engineering (2006, 2013), and of the High Assurance Systems Engineering Symposium (2008). She was an associate editor of the IEEE Transactions on Reliability from 2008 to 2016 and is an associate editor for Software Testing, Verification & Reliability since 2012. She is a scientific advisory board member of the International Conference on Human Error, Reliability, Resilience and Performance (2017). She is a Fellow of the IEEE.

Affiliations

Boyuan Li¹ • Xiaoxu Diao¹ • Wei Gao¹ • Carol Smidts¹

Xiaoxu Diao
diao.38@osu.edu

Wei Gao
gao.1579@osu.edu

Carol Smidts
smidts.1@osu.edu

¹ Department of Mechanical and Aerospace Engineering at the Ohio State University, Columbus, OH, USA