View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Hosei University Repository

法政大学学術機関リポジトリ
HOSEI UNIVERSITY REPOSITORY

A Software Tool to Support Scenario-Based Formal Specification for Error Prevention

著者	Li Siyuan
出版者	法政大学大学院情報科学研究科
journal or publication title	法政大学大学院紀要. 情報科学研究科編
volume	13
page range	1-6
year	2017-03-31
URL	http://doi.org/10.15002/00021516

A Software Tool to Support Scenario-Based Formal Specification for Error Prevention

Li Siyuan
Graduate School of Computer and Information Sciences
Hosei University
3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan
E-mail: siyuan.li.83@stu.hosei.ac.jp

***Abstract*—Formal specification can be an error-prone process for complex systems and how to efficiently write correct specifications is still a challenge for practitioners in industry. This paper presents a software tool to support the scenario-based formal specification approach developed in the SOFL formal engineering method. Using the tool, some suggestion of the further contents of the specification may be automatically predicated to facilitate the user in completing the specification. To improve the readability of the formal specification, the tool can also automatically translate the textual format of the specification into a comprehensible tabular format. Both of these functions can be helpful to prevent errors during the construction of the specification. We discuss each of the functions by first presenting its principle and then illustrating it with examples. We present a case study to show how the tool supports the scenario-based specification approach. Finally, we conclude the paper and suggest topics for future research.**

Keywords—SOFL; Formal specification; Verification; Error prevention

I. INTRODUCTION

As a formal engineering method for practical software development, SOFL has provided a method for functional scenario-based inspection and testing of programs, respectively [1]. The same concept has also been found to be effective in helping construct formal specifications in practice [2] and verify their properties such as consistency [3] and completeness [4].

However, writing a formal specification for a complex system tends to be error-prone. This problem can be attributed to three factors. The first is that the practitioner who writes the formal specification may lack competence of commanding the formal notation. The second is that the habit of writing code may affect the practitioner in keeping the logical consistency and completing the definition of the functionality. For example, when defining an update of a composite object, only some fields of the object are defined in the post-condition while the other fields are undefined. This style may have no problem for programming but usually results in incomplete definition of the functionality in the specification. The final factor has something to do with inappropriate input or output variables and their types. Choosing inappropriate variables and types may lead to errors in specification in most cases.

To deal with the above challenge, we believe that dynamically checking the consistency of the current version of the specification and predicating the further necessary contents of the specification as it is being constructed are an efficient way to build correct formal specifications. To realize this goal, obviously a software tool needs to be developed to support the process of constructing a specification. In our work, we build some principles to support the functional scenario-based formal specification approach based on the SOFL specification language. Specifically, the supporting tool includes following specific functions: (1) automatically checking and predicating necessary contents for the specification, and (2) automatically translating the textual format of the specification into a comprehensible tabular format. We have also developed a tool to implement the principles. All of these can be helpful to prevent errors during the construction of the specification. The result of this research is expected to make the SOFL method more effective and practical in industry where the current SOFL technology has been tested or applied in realistic systems development.

II. INTRODUCTION TO SCENARIO-BASED FORMAL SPECIFICATION

In this section, we briefly introduce the essential idea of the scenario-based formal specification for a process using the SOFL specification language [5]. To this end, we first need to explain the concepts of process and functional scenario, and then illustrate the scenario-based approach to formal specification with an example.

A. Process

A process is the essential component of a module in a SOFL formal specification. Its specification is composed of the signature, pre- and post-conditions. The signature shows the process name, input variables, output variables, and external variables. The pre-condition sets a restriction on the input of the process while the post-condition defines the relation between the input and the output that must be satisfied after the execution of the process. Below shows the general structure of a process specification.

Supervisor: Shaoying Liu

```
process Operate( x:real ) message:string
ext wr y:real
pre true
post  x > 0 and y = 1 and message = "result1" or
      x = 0 and y = 0 and message = "result2" or
      x < 0 and y = -1 and message = "result3"
end_process
```

All of the types used to declare the input, output, and external variables in the process must be clearly defined in the type section of the related module. A module is a mechanism for defining a sub-system by describing its architecture using a condition data flow diagram (CDFD) and specifying the functionality of every process occurring in the CDFD. It also allows necessary constant identifiers, type identifiers, store variables, type and store invariants to be defined properly, which can be used in process specifications.

B. Functional Scenario in Process Specification

As mentioned in the Introduction, a process specification can be effectively constructed by building the disjunction of functional scenarios. A functional scenario is a conjunction of the pre-condition, a guard condition, and a defining condition. The guard condition is defined as part of the post-condition and is characterized by including merely input variables. The defining condition is also part of the post-condition but for defining output variables in terms of their relation with input variables.

For example, below shows a process Test:

```
process Test( x:bool, y:real ) z:string
ext rd w:bool
pre w = true
post x = true and y < 0 and z = "output_1" or
x = true and y = 0 and z = "output_2" or
x = true and y > 0 and z = "output_3" or
x = false and y < 0 and z = "output_4" or
x = false and y = 0 and z = "output_5" or
x = false and y > 0 and z = "output_6"
end_process
```

In this specification, the post-condition is given as a disjunction of six "functional scenarios". Note that by "functional scenario" here, we actually mean the conjunction of the guard condition and the defining condition, omitting the pre-condition of the process in the original concept of functional scenario, because it applies to every such a conjunction in the post-condition. For example, x=true and y<0 and z=output_1 is a "functional scenarios" whose guard condition is x=true and y<0 and defining condition is z=output_1. The original functional scenario should be w = true and x=true and y<0 and z=output_1. By scenario-based formal specification, we mean that the post-condition of a process should be written by gradually adding functional scenarios one by one. The above process Test can be completely by gradually adding more functional scenarios [10].

III. PRINCIPLES

A. Basic Principles

The set of scenarios is the base of our work. Based on the experience of research and the using of SOFL in the past, when we deal with the process part, it will spend some time and generate some errors. It's one of the reasons why we focus on the scenarios of the process. Based on the scenario, we can further analyze the process. We set each scenario as the most basic unit. In this rule, each value of each condition will lead to a different scenario. For example, a process named Process_vars includes three variables, var_a, var_b and var_c. At here we assume all of them are a part of guard conditions in scenarios. We can divide all variables into three types. Every type represent a kind of method to deal with relevant variables [6]. After we make sure their definition, we can further make assumptions.

There is a variable var_a that can be used to constitute guard condition. We can denote this guard condition as gc_a, and this guard condition only includes one predicate. Generally, the expression of this predicate only has one result. But it will lead to a potential problem. This expression has the possibility of failure. So, the guard condition gc_a has two different values in fact. They are success (denote as true) and failure (denote as false). Now, we can denote this type variable like var_a as potential boolean variable. Accordingly, there are potential boolean guard condition and potential boolean guard condition predicate.

Variable var_b is a boolean type (A kind of basic data types in SOFL) variable, it has two value: true and false. Of course, the guard condition gc_b (only includes one predicate) made by var_b have two results: true and false. We can denote var_b as boolean variable. Accordingly, there are boolean guard condition and boolean guard condition predicate.

Variable var_c is a non-boolean variable. For example, var_c is an integer. By mathematics, the number of values of integer var_c is infinite. But when people design the software or write codes, the number of values of var_c is finite. For example, there is a function. It used to judge whether a divisor is 0. Actually, we just need to know whether it is equal to 0. It is same that this divisor is equal to 1 or 2 or other values for this function. So, var_c is denoted as finite-value-non-boolean variable. Accordingly, there are finite-value-non-boolean guard condition and finite-value non-boolean guard condition predicate.

Through the previous discussion, we can adopt a appropriate method to get necessary information, then we can build the set of scenarios. The set of scenarios is a crucial gist of the applicable development. The number of variables and the value range of variable will directly affect the set of scenarios.

For the convenience of writing and reading, SOFL provides many methods to simplify the writing of post-condition. Using the writing rules of SOFL reasonably, when people write the process, many scenarios will not be written as their original expression. In this way, many context can be simplified. Usually, a statement block can replace many scenarios. But in our analysis, we should analyze all original scenarios and list them all. The different writing method does not change the actual quantity. And for this work, we can use the past research results. So there is no more statements about this content.

As for the number of variables and the value range of variables, we can calculate the number of scenarios directly. Let's discuss the above example again. We denote a predicate of guard condition that made by var_a as gcp_a (same as var_b and var_c). At here, we can know gcp_a has one value (if we don't consider the failure of gcp_a), gcp_b has two value, and we rule gcp_c has three value. According to the principle of mathematical, the number of scenarios should be 1*2*3 = 6. Generally, 6 is the biggest number of scenarios as usual. Then, we can get the biggest set of scenarios. In this method, we do not consider the pre-condition in the set of scenarios. Because in a given process, every scenario acquiescently includes the pre-condition [7].

But, this set isn't the final set. For finite-value-non-boolean guard condition (or finite-value non-boolean guard condition predicate), the difficulty is how to set the number of values. There is a solution, we can build the set of scenarios dynamically. Following is the building process of the set of scenarios. First, when we get a variable that be used to build a finite-value-non-boolean guard condition, we set it has only one value temporarily. Second, we build the set of scenarios based on this variable and others temporarily. Third, when the user inputs conditions about this variable, if the user inputs a new value, we can expand the set of scenarios based on the change of the number of values. In this way, we can expand it clearly.

There is an example. At first, we only have 2 variables x and y. x is a boolean type variable (x has two different value), y is a real type variable. If we don't give the value range of y. In this case, y only have one type of value. The number of scenarios is 2 * 1 = 2. Then the user write an expression of predicate $y < 0$. It means y has two type of value. In other words, there are two value ranges of y, $y < 0$ and $y >= 0$. Then we can get the number of scenarios again. It is 2 * 2 = 4 scenarios.

From above analysis process, some additional information can be gotten. We can know the classification of variables and the value range of variables. These variables must be converted to guard conditions or defining conditions. Based on whether the value of these conditions will affect the building of scenarios, we can further divide or merge scenarios. Finally, through all these work, we can build a set of scenarios. It can be used in the realization of the major functions of this tool.

B. Automatic checking the internal consistency

When we get some conditions, we can check the errors of grammar easily. But for the logical errors, it's difficult. The main reason is the cause of the previous method that has no gist of the whole process. But now, the set of scenarios can clearly reflect it. When the user inputs a logical wrong condition, it will be checked based on the set of scenarios. According to different situations, there is a general statement to explain it for errors.

In this process, we can meet following situations.

Case 1. If an error exists in a guard condition. A guard condition predicate must be inconsistent with any guard condition predicates of the scenario. For a guard condition that is composed of different types of variable, we must analyze them respectively.

1) Potential boolean variables. According to the previous discussion, when we build the set of scenarios, the predicate that represent the failure of this expression of a potential boolean variable will not be generated. If a mismatching appear at here, we will build a new scenario for this situation. Then add it into the set of scenarios.

2) Boolean variables. This situation will not happen. Because according to analysis process above, we can cover any situation of a boolean variable.

3) Finite-value-non-boolean variables. The error is caused by the value range. For one variable, two value ranges might be overlap in different expressions.

a) Value ranges are same. This situation will not happen. Because it must accord with one scenario.

b) For one variable, different predicates' value range have same overlapping part. Based on the rule of division about the value range of this type variable, we can divide it again to make it more accurate. It means the number of scenarios will be bigger.

Case 2. If an error exists in a defining condition. It must be self-contradiction. For example, there is a statement, $x<0$ and $x>0$. In fact, we cannot find any value of x satisfies both $x<0$ and $x>0$.

Case 3. If an error exists in a scenario, especially this error will affect the logic between many scenarios. And this error will appear in guard condition and defining condition for a scenario.

1) If it appear in guard condition. The statement is above.

2) If it appear in defining condition. It means different scenarios have the same input and the different output. Actually, it isn't an error, because this situation is allowed in SOFL.

In brief, there are three logical error or checking point, (1) the failure of expression of a potential boolean condition variable. (2) For one variable, different predicates' value range have same overlapping part. (3) The self-contradiction in defining condition. The generation of logical error is mismatch between the text and the set of scenarios. As long as we modify them, the mistake will be solved. And in some situations, it also can be used in update the set of scenarios.

According to above discussion, these result can be used in checking errors. The tool can give the user feedback message when the user writes an error at the appropriate time. It is the way that we check the internal consistency.

C. Automatic Predicating Specification Contents

When user writes scenarios, the expectation can be shown to users. The information of expectation comes from the set of scenarios. We can follow this method. When the user writes conditions for a given scenario, the shown information is the rest part of this scenario. When the user has finished a scenario, the information of other scenarios will be shown to the user from the set of scenarios. The tool only need to give the user appropriate part of the set of scenarios.

D. Automatic Translating Textual Specification to Tabular Form

The construction of tabular form is scenario-based. It is a new expression of text. One tabular form represent at least one scenario. Table I is an example of the tabular form of a scenario. This tabular form developed by the decision table. It includes 2 parts mainly, guard condition part and defining condition part. In this tabular form, these row represent GC and DC are contents of guard condition and contents of defining condition respectively. And if two different guard conditions have one same defining condition. It means this tabular form includes two basal scenario from the set of scenarios.

TABLE I. THE TABULAR FORM OF SCENARIO

Functional Scenario		
GC	x = true	✓
	y = 0	✓
DC	z = 1	✓

There are several steps to build tabular forms from the text representation of scenarios. It also includes some actions about the feedback message.

Step1,

(1) Get all related information of variables.

(2) Build the set of scenarios.

Step2,

(1) Divide scenarios and conditions based on keywords, such as “and”, “or”.

(2) Find all guard conditions and all defining conditions in each scenario.

(3) Find the map between scenarios that the user write and the set of scenario. If the map can be found successfully, it means the set of scenarios includes it. As for other scenarios that has no map, it must need to be modified. Then we record necessary information that can be used in expand the set of scenario. Like the new value of a variable. By the way, the scenario that only have different express method (like “not x=true” and “x=false”), don’t modify it to keep consistent with the scenario that belongs to the set of scenarios. The difference should be recorded.

Step3,

1) Draw tabular forms based on all marked scenarios. And show tabular forms to the user.

2) Finally, give the feedback message to the user.

IV. Software Tool

We have developed a prototype tool to support the scenario-based formal specification approach. In this section [9], we will discuss these functions from the principles at first, then the tool will be introduced.

A. Implementation Structure

The techniques for realizing the important functions have been implemented to certain extent in our tool. The name of this tool is Scenario-based SOFL Writing Supported Tool. Fig. 1 shows the main GUI of this tool. This tool also includes conventional functions. Such as open file, save file, edit text and so on. It mainly includes three area, the management area (left part), the operating area (middle part) the feedback area (right part). The management area can show the structure of a module and the content of a module. The user can select a process, then it will be shown at the operating area. At operating area, the user can edit the process. At feedback area. The user can select shown information. Such as the set of scenarios, the feedback message of the process. Tabular forms also can be shown at this area.

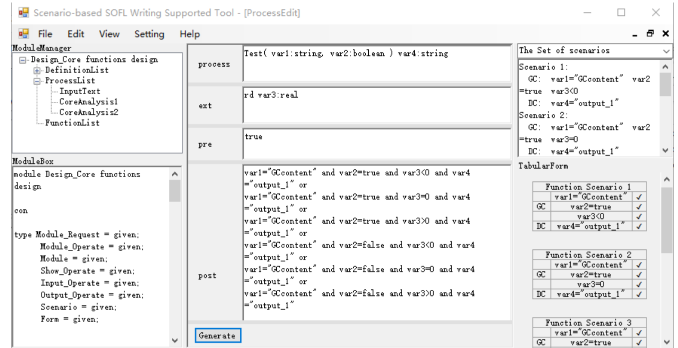

Fig. 1. The UI of the tool.

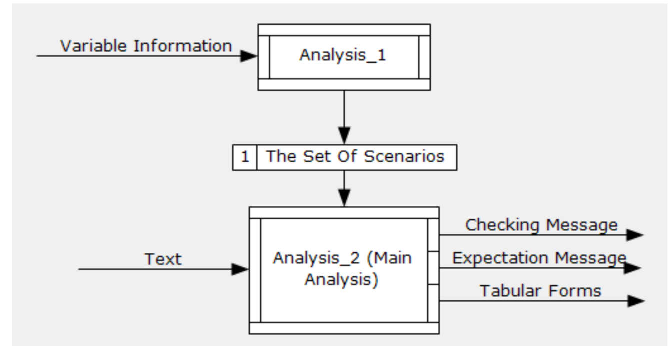

Fig. 2. The CDFD of whole core analysis process.

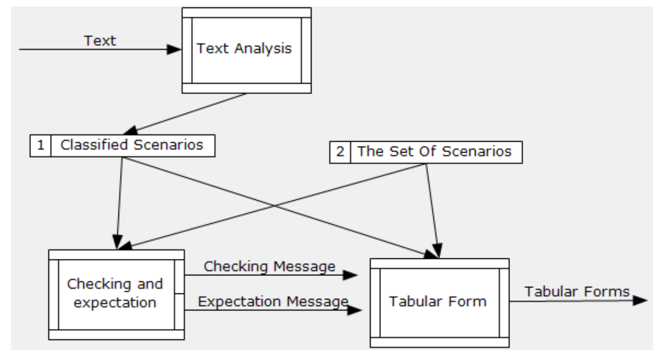

Fig. 3. The CDFD of main analysis.

Both Fig. 3 and Fig. 4 give the CDFD of the core code of the tool to illustrate its overall functions. We can clearly see the process of two core analysis for the functions. From the workflow of the whole process, we can see the set of scenarios is very important. It is the prerequisite of the main analysis for checking and expectation. And it also can be used in the building

of the tabular form to verify it. The analysis process mainly includes four steps below.

Step1. The tool will get necessary information to build initial the set of scenarios. This information only refers to variable information.

Step2. The tool read original text, then analyze it. Next, it will be converted to classified scenarios that can be used more easily.

Step3. Based on the initial set of scenarios, analyze classified scenarios. Then give the user necessary feedback message. It includes the checking message and expectation message. And we can also base on it to update the set of scenarios.

Step4. According to classified scenarios, the tool can show the tabular forms to the user.

B. Case Study

There is a general case, we will use the tool to write a process. Most of situation that might be appeared in the writing will be shown. And the performance of the major functions will clearly reflect how the tool works [8, 10].

Now, let's see this process:

process Test(var1:string, var2:bool) var4:string
ext rd var3:real
pre true
post ......
end_process

TABLE II. VARIABLES

Variable Name	Variable Type	Part Type	Condition Type
var1	string	potential boolean variable	Guard Condition
var2	bool	boolean variable	Guard Condition
var3	real	finite-value-non-boolean variable	Guard Condition
var4	string	output variable	Defining Condition

In this example, it includes 4 variables: var1, var2, var3, var4. From the first line and second line, we can see var1 and var2 are input variables, var3 is external variable (At here, var3 is also an input variable). var4 is output variable. As shown in Table II.

Before the user write the post of the process, the tool can get all information of variables, and build the set of scenarios. Generally, it only be used by the tool itself. There are some table to show how the tool works.

From the Table III, we can see at the right side of the table, every scenario has been shown to the user. At first, the tool can only determine the value of potential boolean variables and boolean variables. As for finite-value-non-boolean variable, we set it only have one value originally. So only 2 scenarios can be shown. And it's a simplified expression, By the way, every scenario must contain the pre-condition. So, there is no need to add the pre-condition into scenarios. In this table, C (var) means a condition satisfied by var. Then, the tool can modify the set of scenarios dynamically, and based on it to give the necessary feedback messages to the user for error prevention.

TABLE III. BEFORE THE USER WRITE THE POST-CONDITION

<table>
<tr><th colspan="2">Existing Information</th><th>The Set of Scenarios</th></tr>
<tr><th>Name</th><th>Type</th><td rowspan="5">Scenario 1:
GC: C(var1), var2=true, C(var3)
DC: C(var4)

Scenario 2:
GC: C(var1), var2=false, C(var3)
DC: C(var4)</td></tr>
<tr><td>var1</td><td>string</td></tr>
<tr><td>var2</td><td>bool</td></tr>
<tr><td>var3</td><td>real</td></tr>
<tr><td>var4</td><td>string</td></tr>
</table>

We do not change variables. If the user writes a condition with logical errors, it can be checked, then the feedback message also can be shown. Through above research, there are three logical errors or checking point. At here, we can write wrong post-condition temporarily to show the first error or checking point, as shown in Table IV. When the var1="GCcontent" is failure, the tool will update the set of scenarios. It only add a new scenario that includes the failure of var1="GCcontent".

TABLE IV. THE FAILURE OF POTENTIAL BOOLEAN GUARD CONDITIONS

<table>
<tr><th colspan="2">Item</th><th>The Set of Scenarios</th></tr>
<tr><th>Pre-condition</th><td>true</td><td rowspan="2">Scenario 1:
GC: var1="GCcontent", var2=true, C(var3)
DC: C(var4)

Scenario 2:
GC: not var1="GCcontent", var2=false, C(var3)
DC: C(var4)

Scenario 3:
GC: var1="GCcontent", var2=false, C(var3)
DC: C(var4)</td></tr>
<tr><th>Post-condition</th><td>not var1="GCcontent" and var2=true</td></tr>
</table>

The second error is the different value range in finite-value-non-boolean condition predicates. We don't specify the value of var3 at the beginning. So when the user adds a condition about var3 and set the value of var3, it means a new value has been generated, and a mismatching appear. So we need to modify the set of scenarios. The new set of scenarios as shown in Table V.

The third error, the self-contradiction of defining condition. The tool will not build new scenario in the set of scenarios. Because, the defining condition can't affect the structure of the set of scenario. It only represents the result of a scenario. So, we only need to check the correction, then modify it.

TABLE V. THE CHANGE OF VALUE RANGE

Post-condition	var1="GCcontent" and var2=true and var3<=0
The Set of Scenarios	Scenario 1: GC: var1="GCcontent", var2=true, var3<=0 DC: C(var4) Scenario 2: GC: var1="GCcontent", var2=true, var3>0 DC: C(var4) Scenario 3: GC: var1="GCcontent", var2=false, var3<=0 DC: C(var4) Scenario 4: GC: var1="GCcontent", var2=false, var3>0 DC: C(var4)

There is a fallible situation, different scenarios have the same input and the different output. It is a logical contradiction. Only need to tell the user the relevant feedback message. Then the user decides how to modify the output.

For the prediction, when the user write conditions for a given scenario, the rest part of this scenario will be shown from the set of scenarios.

Functional Scenario 1		
	var1="GCcontent"	✓
GC	var2=true	✓
	var3<0	✓
DC	var4="output_1"	✓

Functional Scenario 4		
	var1="GCcontent"	✓
GC	var2=false	✓
	var3<0	✓
DC	var4="output_4"	✓

Functional Scenario 2		
	var1="GCcontent"	✓
GC	var2=true	✓
	var3=0	✓
DC	var4="output_2"	✓

Functional Scenario 5		
	var1="GCcontent"	✓
GC	var2=false	✓
	var3=0	✓
DC	var4="output_5"	✓

Functional Scenario 3		
	var1="GCcontent"	✓
GC	var2=true	✓
	var3>0	✓
DC	var4="output_3"	✓

Functional Scenario 6		
	var1="GCcontent"	✓
GC	var2=false	✓
	var3>0	✓
DC	var4="output_6"	✓

Fig. 4. The tabular forms.

At last, tabular forms should be shown to the user. In fact, the generation of tabular forms is real-time. When a scenarios has been written, the tool will show all of scenarios with relevant tabular form to the user. We can see the Fig. 4, it show all of scenarios to the user. In this method, any scenario will be very clear.

V. DISCUSSION AND CONCLUSION

This paper describes the development of SOFL in writing process. We have adopted a new research method -- scenario-based. Using scenario-based research method has many advantages. The biggest one is every scenario is the fundamental functional unit of a process. The detailed study of the scenario is helpful to catch the structure of the whole of process. In this way, whatever writing, reading or modifying, we can do them better. And it is a new research idea on SOFL. We can follow this idea to make more deeply research on SOFL.

Through the whole paper, we can see all of the results of research need the set of scenarios. This paper gives a systematic way to build the set of scenarios dynamically. This is another innovation. Based on the set of scenarios, we can do others easily. According to all of research, we can get the target of error prevention by the improvement of writing method of SOFL.

As for this tool, it shows the main idea of the research, and realize the writing of SOFL. In the process of writing, this tool can show the proper feedback message to help the user. Moreover, it can generate tabular forms of scenario in real-time. This new expression of scenarios can greatly improve reading efficiency, then reduce subjective errors. But our tool also have some shortcomings. Although it can be used in the process of writing of SOFL, but it haven't enough functions as a formal software. Generally speaking, this tool need more improvement.

REFERENCES

[1] Shaoying Liu, Yuting Chen, Fumiko Nagoya, John McDermid, "Formal Specification-Based Inspection for Verification of Programs", IEEE Transactions on Software Engineering, Vol. 35, No. 8, 2012, pp. 1100-1122.

[2] Juan Luo, Shaoying Liu, Yanqin Wang, Tingliang Zhou, "Applying SOFL to a Railway Interlocking System in Industry", Proceedings of the 6th International Workshop on SOFL + MSVL (SOFL+MSVL 2016), LNCS, Springer, Tokyo, Japan, November 15, 2016.

[3] Shaoying Liu, John McDermid, and Yuting Chen, ``A Rigorous Method for Inspection of Model-Based Formal Specifications", IEEE Transactions on Reliability, IEEE Press, Vol. 59, No. 4, December, 2010, pp. 667-684.

[4] Shaoying Liu, "Capturing Complete and Accurate Requirements by Refinement", Proceedings of 8th IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society Press, Greenbelt, Maryland, USA, December 2-4, 2002.

[5] Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Spinger, Heidelberg(2004).

[6] Martin S., Rudy S., Josep H.: Enhanced Latent Semantic Analysis by considering mistyped words in automated essay scoring. Informatics and Computing (ICIC), Mataram, Indonesia (2017).

[7] Senem K., Bahar K., Tank K.: Attribute value-range detection in identification of paraphrase sentence pairs. Signal Processing and Communication Application Conference (SIU). Zonguldak, Turkey (2016).

[8] Liu, S., NaKajima S.: A Decompositional Approach to Automatice Test Case Generation Based on Formal Specifications. Secure Software Integration and Reliability Improvement, Singapore (2010).

[9] Chen H., Shen Y., Jiang J.: Extended SOFL features for the modeling of middleware-based transaction management. Engineering of Complex Computer Systems (ICECCS), Shanghai, China (2005).

[10] Li M., Liu S.: Automated Functional Scenarios-Based Formal Specification Animation. Software Engineering Conference (APSEC), Hong Kong, China (2012)