Automatic Transformation from SOFL Formal Specifications to Functional Scenario Forms for Verification and Validation

Yan, Ye

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
11
(開始ページ / Start Page)
1
(終了ページ / End Page)
6
(発行年 / Year)
2016-03-24
(URL)
https://doi.org/10.15002/00012870

Automatic Transformation from SOFL Formal Specifications to Functional Scenario Forms for Verification and Validation

Yan Ye
Graduate School of Computer and Information Sciences
Hosei University
Tokyo, Japan
Ye.yan.75@stu.hosei.ac.jp

Abstract— Specification-based testing and inspection are two important techniques in the SOFL method for verifying programs, but both of them are established on the basis of the concept known as functional scenario form (FSF). In this paper, we describe how a SOFL formal specification can be automatically transformed into a FSF. The transformation is realized in four steps: lexical analysis of the formal specification, conversion from the specification to Reverse Polish Notation (RPN), transformation from RPN to Disjunctive Normal Form (DNF), and derivation of a FSF from the DNF. Our discussion focuses on the first three steps that have already been realized, but we will also discuss how an existing algorithm can be used for the conversion from the DNF to a FSF for verification and validation. We present the related algorithms and illustrate them with examples. Finally, we evaluate our algorithms implemented in the tool by testing.

Keywords— SOFL specification, Lexical Analyzer, RPN, DNF, FSF, verification and validation

I. INTRODUCTION

With the rapid growth of large software systems, formal methods become more and more important in development activities as Fig.1 shows, the formal specification language can generate precise and unambiguous requirements document to provide a rigorous mathematical foundation for software projects. However, it also has limitation, such as high demand to developers, only providing symbols and rules, poor readability of specification etc. In addition, as the software activities becoming more complex and large-scale, applying formal methods becomes more difficult and expensive than before in realistic projects.

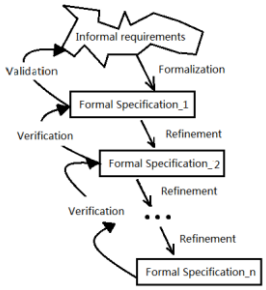

Fig.1. SOFL application in software activities

In order to improve the above situations, SOFL, resulting from an integration of VDM with DFD, was designed as a formal engineering solution. SOFL provides a simple and reliable platform that is suitable for specification-based development. It is not only providing symbols and rules for specification constructing, but also equipped with related techniques for various software activities. SOFL creates a rigorous, efficient and structured method. Its practicability can be said superior to existing formal methods [1].

Now, automatic transformation from SOFL formal specifications to a functional scenario form (FSF) becomes an important technique to support specification-based applications for programs in SOFL method [2]. Compared with manual transformation, automatic transformation can significantly save time, mitigate the workload and reduce chances for committing mistakes. It establishes a "mapping" relationship between SOFL formal specification and executable program to facilitate process revision, adjustment and testing. Moreover, its intermediate process has the characteristics of RPN and conduces to the mechanical implementation on one side. On the other side, automatic transformation makes a formal specification split into many relevant items that carry messages, so its content and form could be converted flexibly to provide an appropriate interface for multiple applications like test data generation etc.

The rest of the paper is organized as follows. Section II shows the related work. Section III shows some concepts to help readers understand the automatic transformation works. Section IV briefly introduces the whole implementation in automatic transformation. Section V cites the cases to prove the techniques over the whole process. Subsequently, Section VI gives the analysis and evaluation of the results in each part. At last, section VII makes a conclusion of the current work and envisages future work.

II. RELATED WORK

In the paper titled "An Approach to Transforming Visual Formal Specifications to Java Programs" [3], Liu has discussed how to transform the CDFDs into executable java program. It describes the policies and rules in transformation process of various CDFD structures. The author proposed a series solutions based on Morgan's refinement rules [4], so the transformed results meet the functional requirements of CDFD. The proposed policy is used to transform the CDFDs into java

framework and the rules are used to transform various structures of CDFD, but it didn't give a realization. However, the idea of this article makes basis of automatic transformation of SOFL specification.

The Chapter 19 of the book "Formal Engineering for Industrial Software Development" [5] have comprehensively discussed the conversion from the data type, class, model to the program, it also covers some special structure in SOFL, such as multiple interfaces etc. It is mainly discussed the possibility of SOFL automatic converting from interface arrangement, data structures and some other parts.

In another article "Automatic Transformation from Formal Specifications to Functional Scenario Forms for Automatic Test Case Generation" [6], the authors further proposed some specific conversion methods. It firstly discusses the transformation of complex expressions, quantization expressions and equivalent expressions, then showing how to convert from the predicate expression to generate automatic test cases in detail and provides an example. This paper is inspired and worth to learn, such as how to deal with the branch case. However, because this article is for VDM-SL, the operator and definition are come from VDM-SL and it didn't address the issue in SOFL. Besides, some algorithms theoretically use the logic symbol which is less suitable for nesting and some other cases, so these parts should be improved in practical perspective.

Another paper [7] proposed a formal automatic test technology. This technique uses the specified black box test cases group which includes the universal quantifier and existential quantifier of specification to generate logic diagrams (e.g. testing framework) automatically. Unlike processing the language directly in tradition, it dealing with quantifier straight, so it can be more widely applied to the expressive specification. To handle with the quantifiers (e.g. "exsit") is necessary in our research, this idea is worth to learn.

III. THE RELATED CONCEPTS AND PROBLEMS

In this paper, we describe all of the major techniques necessary for the conversion from a SOFL specification to FSF. The entire process is illustrated in Fig.2. In order to achieve this goal, the first step is to design a Lexical Analyzer (LA), it could identify and process all kinds words in SOFL. Then, SOFL specification could be converted into the RPN by the improved RPA based on the LA result, optimized data structures and storage structure. Next, we build a bridge between RPN and DNF to obtain the prerequisite for FSF which is still expressed in RPN. Finally, we show how this DNF be changed to the FSF for verification and validation.

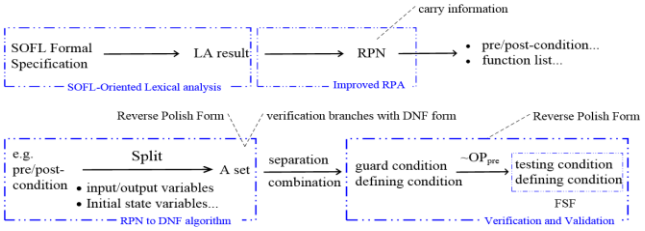

Fig.2. The automatic transformation process

A. *Pretreatment work*

The preparation work is implemented to lay a good foundation and guarantee the effective execution for the follow-up works.

Firstly, we should extract pre/post-condition out from SOFLTOOL (a supporting tool of SOFL) and record all kind variables and functions that are needed in verification and validation. Because pre/post-condition is stored as an XML format in SOFLTOOL, we store it into a XML file to keep its integrity and originality. Every part of specification has its fix position in this intermediate file. That is to say, we should be familiar with the SOFL specification structure as Fig. 3 shows:

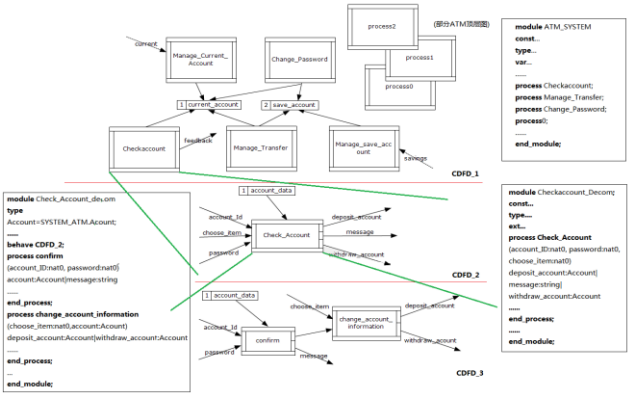

Fig.3. SOFL structure and hierarchical

Then, both keywords list and the corresponding priority list should be arranged. The former affects whether LA could recognize words correctly and the latter determines whether the order is accurate in RPN, We must guarantee the result is complete without omission.

Thirdly, we should be familiar with the data and symbol type, then considering about the data structure and storage mode to facilitate the subsequent operations.

B. *Lexical Analysis*

After the necessary pre-processing work, we could start to do the body work. Lexical analysis is the first step in our research and plays an important role. It reads text from the source file and decomposing it into a series of words based on the lexical rule. The word is the basic symbol of program language. The lexical rule is different according to the different source languages, so the output result is not the same. Fig.4 shows LA mechanism, we generally abstract the grammar from the source language and form the regular expression, NFA, DFA etc. Then, we get the state transition diagram (STD). Finally, we match the words with the lexical rules and get the LA result. In this part, the issues should be solved as follows:

- To distinguish the multiple end situations in extracted file;
- To apply the new keywords list of SOFL to the LA;
- Extract and register functions and all kind variables;
- To mark and maintain the related information at this stage;
- Clearly understand the word formation rules of SOFL, then to form the DFA, NFA etc. finally, to get the STD for programming;
- Compared to the traditional LA, SOFL needs to manage additional situations.

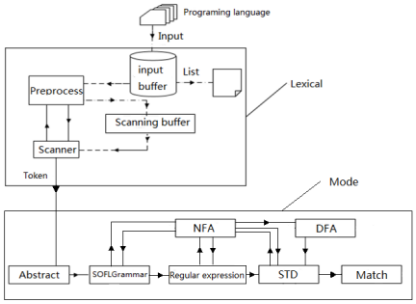

Fig.4. Intrinsic mechanism of the lexical analysis

C. *Reverse Polish Algorithm*

In the automatic conversion work, LA and RPN are tightly linked. LA is the basis of RPN and the output of LA is the input of RPN.

RPN is also called suffix expression. As we all know, the general expression whose binary operator is always located between its two parameters is called infix expression. RPA could convert an infix expression into a suffix expression. The feature of it is that each operator is located behind its operands after transformation and the relative position of operands keeps fixed which could be seen in Fig.5. The calculation of RPN is depends on the order of operand and operator without the brackets, that is very beneficial for mechanical achievement. In this paper, converting the SOFL formal specification into the RPN is a critical intermediate step for automation.

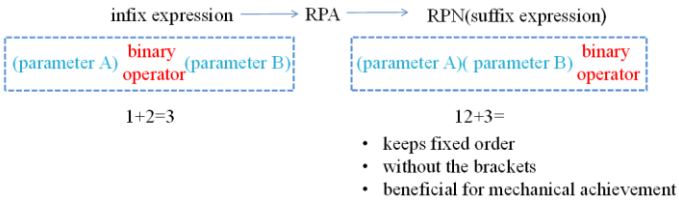

Fig.5. feature of infix expression and suffix expression

Given it is SOFL oriented, the traditional RPA is unable to meet the demand of this study. It needs to be improved in the following aspects:

- The traditional RPA do calculation directly, but here need to combine it with the LA result to get all kind words;
- Traditional RPA only dealing with the arithmetic operators and simple operation. But the function name, logical operators etc. also need to be handled in SOFL;
- Compared to the traditional RPA, it not only need to know the type, priority level and process mode of each word, but also need to record as much as possible messages, outputting the corresponding RPN results while at the end of pre/ post-condition etc. in order to serve verification and validation;
- To guarantee the pre/post-condition is a whole expression, so how to handle the branch case should be focused on.

D. *Functional Scenarios Form for verification and validation*

FSF is already defined in the previous publication by Liu's paper [6], it is a key concept for verification and validation. As Fig. 2 shows, the FSF of SOFL specification originates from three parts: pre/post-condition, input/output variables and ext part. The FSF consists of three components. One is called guard condition that only contains input and initial state variables, the other is called defining condition which includes at least one output variable or the state variable, the remaining one is initial pre-condition. We also name the format which conjunctive the guard condition with the initial pre-condition as testing condition (~ represent the initial value of variables).

In general, As Fig.6 shows, a FSF is a disjunction of functional scenarios: F1 or F2 or ... or Fn, where each Fn is called a *functional scenario*, which is a conjunction of pre-condition, guard condition and defining condition, its format is as: P1 and P2 and ... and Pm.

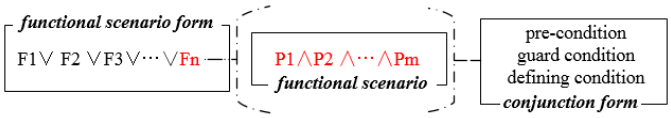

Fig.6. The functional scenario form in general

The premise of forming FSF is to get the DNF of post-condition [5], and this can be done by using the following three laws: DeMorgan law, distribution and associative laws. After previous work, the post-condition has been changed to the RPN, so we need thinking about how to build a bridge between RPN and DNF.

IV. IMPLEMENTATION

A. *Preparation work*

As mentioned before, we need to do some preparation work before realizing the automatic transformation.

Firstly, we should extract specified SOFL formal specification from SOFLTOOL. Because it is saved as an XML format in SOFLTOOL, we store it into an XML file called SYSTEM.XML to keep its integrity and originality. Every part of SOFL specification has its fix position in this intermediate file. Then, we register all kind variables and functions that are needed in verification and validation. Taking the variables as an example, we choose Hash table as the storage structure and offer two kind formats to register this messages, one is the original format, the other is the code format which is based on the keyword list that aiming to facilitate the operation. Every keyword has one unique and corresponding code (of course, we offer the contrast table between the keyword list and the code).

Secondly, keyword list affects whether LA could correctly recognize words, and the corresponding priority list determines whether the order is accurate in RPN. We should guarantee these lists are complete and no omission. Considering the functionality of them, the query efficient also should be focused on.

Thirdly, we should be familiar with the data and symbol type, then consider about the data structure and storage mode to facilitate the operations and being friendliness.

B. *SOFL- oriented Lexical Analysis*

After finishing preparatory, we can start to build the Lexical Analyzer. Its design emphasis is on the transformation

from Deterministic Finite Automaton (DFA) to the State Transition Diagram (STD) [8] and programming.

We build the STD with four kind states, initial state, intermediate state and final state which are represented by different circles respectively. In order to raise the logic, we specially add the judging state into STD and the judge condition depends on different situations. Here only list one path of STD as Fig.7 due to the limited space.

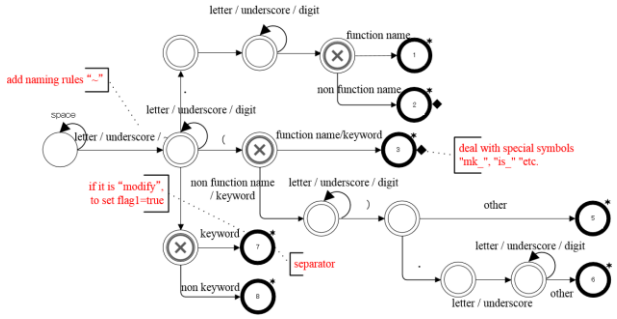

Fig.7. State Transition Diagram of SOFL (part)

Through analysis of LA, we know the below issues are involved in the LA:

- Programming with the STD;
- To distinguish the various end situations (include the pre/post condition, expression etc.);
- To execute the new LA to get the correct result through use new keywords list of SOFL etc.;
- To keep the related information at this stage and add new messages gradually;
- Compared to the traditional LA, SOFL-oriented LA should manage additional cases as in Table 1.

Table 1. Typical cases in SOFL

Case		Example		Description
type	Separate	map to		split by the space
		case…of		split by expression / variable
	Delimiter	,		many situations
		;		three typical cases
	Combine	mk_A		combined operator with operand
	Keyword	~a		not conform to naming rules
		true		special keyword
same start		<	less than	same start but different symbols
		<=	not bigger	
		<>	not equal to	
		<...>	Enumeration	
function		fun . A/ a. fun . A		two typical cases
place	different	=	pre-post/ type	judgment/ renaming
	same	->	modify/others	revise/ mapping

In addition, we adopt advanced search technique which is commonly used in this process. We should focus on the search indicator backtracking, whether it is backtrack, when it is back and what place it should back to.

After finishing these works, we could go on transforming the LA result to the RPN.

C. *Optimize Reverse Polish Algorithm*

Our ultimate goal is to enable RPN that comes from SOFL specification could be transformed to serve verification and validation, so we need to provide information as detailed as possible for converting the RPN to the format that verification and validation needs. The essence is to mark this information and maintain it from the start to the end.

As mentioned before, RPA is used to transform from the infix expression into the suffix expression. In this study, the infix expression is the predict text (includes pre/post-condition, all kind variables and function section etc.) that has extracted from SOFLTOOL and saving it in the intermediate XML file. Then, we should convert the specified part in this text to the RPN with keeping and adding related messages on the basis of LA result. Aiming to solve the problems in Section III.C, there mainly have two aspects should be optimized in RPA. One is to improve the relevant structures, the other is adjusting the algorithm for SOFL. Next, the results should be written into a XML file. We design several storage formats to record different messages according to the characteristic of XML.

Based on the above data and storage structures, we work out the improved RPA flow as Fig.8 shows. It uses the LA result as its input and output the RPN with related messages.

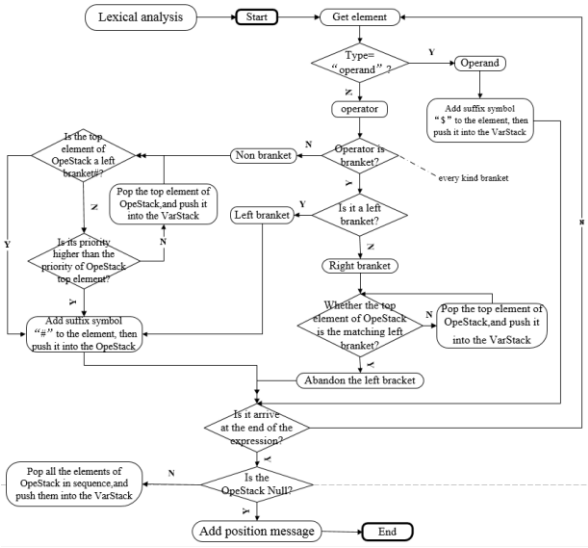

Fig.8. Improved RPA flow

The items in SOFL specification have different reverse polish format, here only show two cases in Table 2:

Table 2. The RPN of some typical cases

Category	Description		Typical example	Reverse Polish Notation
function	function formats		A.fun(a,b)	a b , A.fun
			fun(a,b)	a b , fun
			fun()	fun
	SOFL function operator		modify(A,a->b,c->d)	A a b-> , d e -> , modify
			a inset dom(b)	a b dom inset
			override(a,b)	a b , override
separate symbol	,		multiple situations	be used then ingored
	|		multiple situations	be used then ingored
	;	in if	if a ; b; then c else d	a b and if c then d else
		in case	case a of b->c; d->e	a case b c -> d e -> or
		others	e.g. A; B; in explicit	be ignored directly

D. *From DNF to FSF to Serve verification and validation*

As mentioned before, the premise of forming FSF is to get the DNF of specified specification. Now we have got the pre/post-condition expressed in RPN, so the first step is to change the RPN to DNF as the prerequisite for functional scenario form. By analyzing the feature between RPN and DNF, we figure out an algorithm to convert RPN to DNF. At last, we get the DNF of specification.

Here, owning to Liu's paper [6] has put forward an algorithm to transform from DNF to FSF, so we implement this existing method in our work. Here the DNF is the RPN, so we adjust that method to make it apply to this research. Here give Case 1 to illustrate the FSF forming process.

Case 1. A simple DNF specification (RPN)

```
Process OP (a, b: nat )
Ext  wr  c:real
Pre   c 0 <>
Post  a  ~c > c a b - ~c * = ~c b > ∧∧   (P1)
      a  ~c > c a b + ~c /= ~c b > ∧∧     (P2)
      a  ~c <= c a b / 1+~c / > ∧         (P3)
                ∨∨
```

(1) For P_1, it has three sub conjunctive normal items.

$Q_{1=}$ a ~c >, $Q_{2=}$ c a b - ~c * = and Q3 = ~c b >

(2) To judge each sub conjunctive normal item, if it only includes the input variables (a, b) and initial state variable (~c), then put this item into the Out_1 collection, conversely, if it contains at least one output variable (here has no output variables) or state variable(c), then add it into the Out_2.

(3) We get Out_1 is {a ~c > ~c, ~c b > }, Out_2 is { c a b - ~c * =}

(4) So for the P1, S_1^1 is {a ~c >~c b >∧}, S_2^1 is {c a b - ~c * =}

(5) Repeat the above 2~4 steps, the S of P_2 and P_3 are:

S_1^2 = a ~c >~c b >∧, S_2^2 = c a b + ~c /=

S_1^3 = a ~c < = , S_2^3 = c a b / 1+~c / >

(6) S_1^1 is equal to S_1^2, so merge S_1^1 and S_1^2 and disjunctive their corresponding S_2^1 and S_2^2 as a form like $S_1^1 \wedge (S_2^1 \vee S_2^2)$:

a ~c >~c b >∧ a b - ~c * =c a b + ~c /= ∨

(7) The others guard conditions are different from each other. so conjunctive S_1^i and S_2^i directly to get:

a ~c <= c a b / 1+~c / > ∧

(10) To combined with initial pre-condition ~c 0 <>, to get

1) ~c 0 <> a ~c >~c b >∧ a b - ~c * =c a b + ~c /=∨∧∧

2) ~c 0 <> a ~c <= c a b / 1+~c / >∧∧

(11) Finished, we get the FSF as (10) shows, it has two *functional scenarios* 1) and 2) and each them has its defining condition (green) and testing condition (red).

V. TEST

This section takes a ATM withdraw specification to proof the algorithms in accordance with the transformation process, including SOFL-oriented LA, the optimized RPA, RPN to DNF etc.

A. *The test of LA and RPA*

The Fig.9 includes the various variables, pre/post-condition and method names. While implement the LA and RPA on it, we get the XML form as Fig.10 shows, which shows the result of RPN that carry with information.

Fig.9. SOFL Specification of ATM withdraw (part)

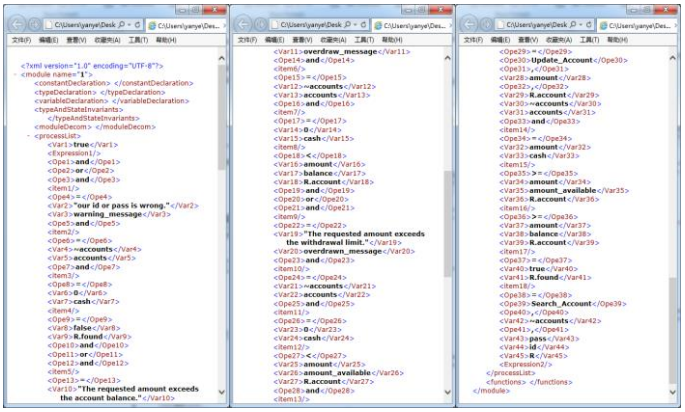

Fig.10. The RPA result of Fig.9

B. *Serve for verification and validation*

In this part, we firstly convert the RPN of Fig.10 to the DNF which is the premise to forming FSF, the result is the form of RPN, but in order to improve the readability, we show the original form of these four disjunctive items in Fig.11.

R_1 = **Search_Account**(id, pass,~accounts) and R_1.found=true and R_1.account.balance >= amount and R_1.account.amount_available >= amount and cash=amount and accounts= **Update_Account**(~accounts,R_1.account,amount)

R_2 = **Search_Account**(id, pass,~accounts) and R_2.found=true and R_2.account.balance >= amount and R_2.account.amount_available < amount and cash=0 and accounts= ~accounts and overdrawn_message="The requested amount exceeds the withdrawal limit."

R_3 = **Search_Account**(id, pass,~accounts) and R_3.found=true and R_3.account.balance < amount and cash=0 and accounts= ~accounts and overdrawn_message="The requested amount exceeds the account balance."

R_4 = **Search_Account**(id, pass,~accounts) and R_4.found=false and cash=0 and accounts=~accounts and warning_message="our id or pass is wrong."

Fig.11. All the items of DNF of case 2

The FSF forming process has introduced in Section IV.D, do like that, we obtain all the functional scenarios in a FSF

derived from the DNF in Fig. 11. Fig.12 shows its infix format. We can see that this withdraw process has four *functional scenarios*, and every disjunctive clause is composed of a testing condition and a defining condition. Then, we could use this FSF to serve verification and validation.

Testing condition: R_1=Search_Account(id, pass, ~accounts) and R_1.found=true and R_1.account.balance>=amount and R_1.account.amount_available>=amount
Defining condition: cash=amount and accounts=Update_Account(~accounts,R.account,amount)

Testing condition: R_2=Search_Account(id, pass, ~accounts) and R_2.found=true and R_2.account.balance >=amount and R_2.account_balance< amount
Defining condition: cash=0 and accounts=~accounts and overdrawn_message="The requested amount exceeds the withdrawal limit."

Testing condition: R_3=Search_Account(id, pass, ~accounts) and R_3.found=true and R_3.account.balance<amount
Defining condition: cash=0 and accounts= ~accounts and overdrawn_message="The requested amount exceeds the account balance."

Testing condition: R_4= Search_Account(id, pass,~accounts) and R_4.found=false
Defining condition: cash=0 and accounts=~accounts and warning_message= "our id or pass is wrong."

Fig.12. The FSF of Fig.9

VI. EVALUATION

Though analyzing the previous test results, the intermediate file has stored the SOFL formal specification that is extracted from SOFLTOOL correctly, and every part of SOFL formal specification has its own appointed position. This file is as the input for LA and the LA result is as the input for improved RPA successfully.

After execution of the lexical analysis and Reverse Polish Algorithm, via analyzing the result shown in Fig.9, we can know from the perspective of operation object, all operands and operators can be accurately identified. Besides the ordinary words, functions like "Search_Acount" also can be identified as an operator. The parameters of these functions are also be separated clearly by delimiter and the order of delimiter and the parameter are correct as well. After adding the naming rules, "~account" as a whole could be correctly identified as an operand. Also, the symbols like ">" have the same start can also be distinguished. Various brackets are ignored. Special keyword "true" is exactly identified as an operand. The Composite type "R.account.amount_available" can also be correctly identified as the operand. The ESC of XML can be automatically converted back as well.

From the view of location, the overall sequence is correct, and the range of every sub disjunctive item is accurately marked. Pre/post-condition has been precisely separated from each other and the corresponding RPN can be output correctly respectively. Finally, the structure of the whole result is consistent with the characteristic of RPN.

After finishing the above work, we go to convert the RPN to the DNF. The transformation result shows it correctly translating the RPN to the DNF. It maintains the integrity and feature of RPN and inherits the related information like type, serial number of RPN result. Moreover, it adds the sign of each sub disjunctive item. This DNF format is the premise to get the FSF for verification and validation.

Ultimately, we translate this DNF to the FSF. The result shows the functional scenarios as expected, and every functional scenario is composed of a testing condition and a defining condition which is corresponding to a disjunctive item in DNF. Of course, this result is the reverse polish format. These conditions are keeping the related messages and abide by the definition of FSF components.

VII. CONCLUSION AND FUTURE WORK

Aiming to achieve automatic transformation from SOFL formal specification to FSF for verification and validation, we have designed a SOFL-oriented Lexical analyzer firstly and optimized the RPA, then making a bridge from RPN to DNF to form the prerequisite for FSF. This means we have set up a foundation for conversion into FSF. Furthermore, in this study, the intermediate process not only has the characteristics of RPN, but also carrying useful messages that is superimposed by different stages in automatic transformation process. In addition to verification and validation, this intermediate result could offer flexible interfaces for some other applications. At last, we use some cases to show the whole processing procedure. It shows that the SOFL formal specification could be successfully transformed to the expected format for verification and validation.

In the future, we will continue to make the full execution automatically, and adjust the intermediate process and interfaces to serve for as much as possible applications. Furthermore, we will also make the SOFL tool to be more comprehensive as well.

REFERENCES

[1] Liu, S., "Formal Engineering for Industrial Software Development", BeiJing: Tsinghua University Press, 2008:7.

[2] Liu, S., Nagoya, F., Chen, Y., Goya, M., & McDermid, J. A. "An automated approach to specification-based program inspection". In Formal Methods and Software Engineering. Springer Berlin Heidelberg. pp. 421-434, 2005

[3] Liu, S., "An Approach to Transforming Visual Formal Specifications to Java Programs", Journal of Three Dimensional Images， 17(1): pp. 121-128, 2003.3.

[4] Carroll Morgan, "Programming from Specifications", UK: Prentice-Hall International Ltd, pp.3-13, 1990.

[5] Liu, S., "Formal Engineering for Industrial Software Development". BeiJing: Tsinghua University Press， pp.349-380, 2008.

[6] Liu, S., Hayashi, T., Takahashi, K., Kimura, K., Nakayama, T., & Nakajima, S. "Automatic transformation from formal specifications to functional scenario forms for automatic test case generation". In Proceedings of the 2010 conference on New Trends in Software Methodologies, Tools and Techniques: Proceedings of the 9th SoMeT_10 , pp. 383-397, IOS Press, August 2010.

[7] Michael R.Donat, Donat M R. TAPSOFT '97: "Theory and Practice of Software Development". Springer Berlin Heidelberg, Volume 1214: pp.833-847, 1997.

[8] Andersen, M., Elmstrøm, R., Lassen, P. B., & Larsen, P. G. "Making specifications executable—using IPTES Meta-IV". Microprocessing and Microprogramming, 35(1), pp. 521-528.,1992.