A Supporting Tool for Syntactic Analysis of SOFL Formal Specifications and Automatic Generation of Functional Scenarios

Zhu, Shenghua

(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)
9

(開始ページ / Start Page)
33

(終了ページ / End Page)
38

(発行年 / Year)
2014-03

(URL)
https://doi.org/10.15002/00010518

A Supporting Tool for Syntactic Analysis of SOFL Formal Specifications and Automatic Generation of Functional Scenarios

Shenghua Zhu
Graduate School of Computer and Information Sciences, Hosei University, Tokyo, Japan
shenghua.zhu.7h@stu.hosei.ac.jp

***Abstract*—SOFL formal specifications have been proved to be useful and expressive enough in describing functional requirements for software development. Based on SOFL formal specifications, many techniques have been proposed to provide us with effective solutions for software verification and validation. To support these techniques, a tool support for analysis of specifications is necessary. However, such a tool is still not available. In this paper, we present our work on a supporting tool. The tool supplies two fundamental functions: syntactic analysis of SOFL formal specifications and automatic generation of functional scenarios. By syntactic analysis, we can get syntactic information of SOFL specifications. The tool can create an xml file for storing and reusing the syntax information. Functional scenarios are well-structured predicate expressions, which could be derived from formal specifications. Many formal specification-based techniques require the generation of functional scenarios. Our tool also supports automatic generation of functional scenarios on the basis of the syntactic information.**

***Keywords*—SOFL, Formal specifications, Syntactic Analysis, Functional scenarios.**

I. INTRODUCTION

SOFL formal specifications generally consist of two parts: modules and corresponding CDFDs (Control Data Flow Diagram). Modules are responsible for precisely defining the requirements and CDFDs provide a graphic explanation of the cooperation of processes in each module. Because SOFL benefits both the advantages of formal notations and graphic expressions, SOFL has the "talent" to describe both functional requirements and the architecture of software. In addition, based on SOFL formal specifications, many techniques have been proposed to provide us with effective solutions for software verification and validation. SOFL could be used practically in real software development.

However, compared with some other formal languages, SOFL is restricted by tool supporting, especially for some fundamental functions. In order to make it more practical, we hence implemented a supporting tool for SOFL. This tool can supply two fundamental functions: syntactic analysis of SOFL formal specifications and automatic generation of functional scenarios. By syntactic analysis, we can obtain syntactic information of SOFL formal specifications. The tool can create an xml file for storing and reusing this syntax information. Functional scenarios are well-structured predicate expressions, which could be derived from formal specifications. Many formal specification based techniques require the generation of functional scenarios [1][2]. In our tool, we also realized automatic generation of functional scenarios on the basis of syntactic information. Our work is expected to be effective in reducing time and budget by a large margin in applying SOFL to a real software development.

The remainder of this paper is organized as follows. Section 2 talks about the background of our work including features of SOFL, definition of functional scenarios and brief introduction to strategy design. Section 3 describes the details of our tool, which includes two core components: a parser for SOFL and a processor for generating functional scenarios. Section 4 mentions some related work. Section 5 makes a conclusion of our current work and points out how it could support future research.

II. BACKGROUND

SOFL is short for Structured Object-based Formal Language, which was firstly proposed in Liu's paper [3]. In order to make it more adaptable for practical software development, the designer considered overall the advantages of formal notation, structured methods and object-oriented method, and successfully found out a complementary approach to integrate these three ideas into one formal language. Formal specifications written in SOFL should be encapsulated in a series of modules. Each module represents a high-level system or low-level sub-system and one module also could be decomposed further to lower-level modules. In each module, we should abstract all involved resources, declare them by classifications and define the main processes (operations) to complete the functionality. For each process, it uses pre-condition to describe the assumed initial state and post-condition to clarify the expected final state. Pre- and post-conditions are common predicate expressions, in which sub-predicate clauses could be connected by logic connectors without no regular pattern.

As mentioned before, SOFL formal specifications are synthesis of many kinds of descriptions, including description for resources' definition and declaration, description for operation, while functional scenarios are predicate expressions which have been well classified and partitioned according to information mainly extracted from operations.

In order to generate functional scenarios from formal specifications, firstly we need to clarify the format of operation specification and the concept of functional scenario. Here for simplicity, we adopt Liu's notation in [4] for operations of SOFL formal specifications.

Definition 1: Let $OP(\mathrm{OP}_{iv};\mathrm{OP}_{ov})[\mathrm{OP}_{pre};\mathrm{OP}_{post}]$ denotes an operation of SOFL formal specifications, in which OP_{iv} represents set of input variables whose values should not be changed by this operation, and OP_{ov} represents set of output variables whose values could be newly produced or updated by this operation, and OP_{pre} OP_{post} represent pre- and post-conditions of respectively.

Then we define functional scenarios based on criterion 1.

Definition 2: Functional scenario is a predicate expression matched with specific pattern. Let $\sim OP_{pre} \wedge C_i \wedge D_i$ denotes one function scenario, and each function scenario serves as one disjunctive clause of the following disjunction: $(\sim OP_{pre} \wedge C_1 \wedge D_1) \vee (\sim OP_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (\sim OP_{pre} \wedge C_n \wedge D_n)$. The disjunction is called a *functional scenario form*. In one functional scenario, C_i is called "guard condition", which contains no output variables and satisfies $C_i \wedge C_j = \mathit{false}(\mathrm{i} \neq \mathrm{j})$; D_i is called "defining condition", which involves at least one output variable, and defines the expected final state of output variables. Guard conditions and defining conditions come from OP_{post}.

The algorithm, for generating functional scenarios from formal specifications, has been discussed in Liu's paper [5]. In order to describe the algorithm, we also need notations below:

$V_{ov}(\mathrm{OP}, \mathrm{E})$: denotes the set of variables from OP_{ov} which occur in the predicate E.

$[1 \ldots \mathrm{n}]$: denotes the set of integers $\{1, 2, \ldots, \mathrm{n}\}$.

$A \equiv B$: means that A is the same with B both syntactically and semantically.

Algorithm:

Step 1: Convert the post-condition OP_{post} to disjunctive normal form: $P_1 \vee P_2 \vee \ldots \vee P_n$, where each $P_t(\mathrm{t} \in [1 \ldots \mathrm{n}])$ is a conjunction of atomic predicates or the negation of atomic predicates. An atomic predicate could be a relation (saying $x > y * 5 + z$), a boolean variable, a truth value, or a strict quantified expression. Here if $OP_{post} \equiv \mathit{true}$ or $OP_{post} \equiv \mathit{false}$, go to step 8; else go to next.

Step 2: For each $P_t \equiv R_1 \wedge R_2 \wedge \ldots \wedge R_m (\mathrm{m} \geq 1)$, construct the partition $\{\mathrm{B}_1, \mathrm{B}_2\}$ for the set $\{\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_m\}$ that satisfies the conditions:

(1) $R_i \in B_1 \Rightarrow V_{oe}(\mathrm{OP}, \mathrm{R}_i) = \varnothing, \mathrm{i} \in [1 \ldots \mathrm{m}]$

(2) $R_i \in B_2 \Rightarrow V_{oe}(\mathrm{OP}, \mathrm{R}_i) \neq \varnothing, \mathrm{i} \in [1 \ldots \mathrm{m}]$

Step 3: For each predicate set B_k where $k \in \{1,2\}$, if $B_k \neq \varnothing$, then form the conjunction $Q_k^t \equiv \wedge_{i \in s} R_i$, where $s = \{\mathrm{i} \in [1 \ldots \mathrm{m}] \mid \mathrm{R}_i \in \mathrm{B}_k\}$; otherwise, let $Q_k^t \equiv \mathit{true}$.

Step 4: Express P_t as the conjunction of every such $Q_k^t : P_t \equiv Q_1^t \wedge Q_2^t$. Here corresponds to the guard condition, which involves no output variables in OP_{ov}. Q_2^t corresponds to the defining condition, which contains at least one output variable.

Step 5: Construct the partition $\{\mathrm{A}_1, \mathrm{A}_2, \ldots, \mathrm{A}_w\}$ for the set $\{\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_n\}$ obtained from No 4 that satisfies the condition: $P_i, P_j \in A_k \Rightarrow Q_1^i = Q_1^j$, assuming $P_i \equiv Q_1^i \wedge Q_2^i$ and $P_j \equiv Q_1^j \wedge Q_2^j$, $i, j \in [1 \ldots \mathrm{n}]$, $\mathrm{k} \in [1 \ldots \mathrm{w}]$.

Step 6: For each A_k, form a predicate $P_{A_k} \equiv {}^{\sim}OP_{pre} \wedge Q_1 \wedge (\vee_{l \in [1 \ldots u]} Q_2^l)$, assuming $P_1, P_2, \ldots, P_u$ are members of A_k, $u \neq n$, and each $P_l \equiv Q_1 \wedge Q_2^l$, where Q_1 is the common guard condition and Q_2^l is a defining condition. The decorated pre-condition ${}^{\sim}OP_{pre} = OP_{pre}[{}^{\sim}\mathrm{x}/\mathrm{x}]$ denotes the predicate by substituting the initial state ${}^{\sim}\mathrm{x}$ for the final state x in OP_{pre}.

Step 7: Form the disjunction $P_{A_1} \vee P_{A_2} \vee \ldots \vee P_{A_w}$, which is the functional scenarios form (FSF) for OP, where each P_{A_k} denotes a functional scenario. Then go to Step 9.

Step 8: Form the conjunction ${}^{\sim}OP_{pre} \wedge OP_{post}$ as the functional scenarios form (FSF) for OP.

Step 9: The end.

III. OUR WORK ON SUPPORTING TOOL

The tool consists of two core components. One is a parser for SOFL formal specifications, and the other is a processor in charge of automatic transformation. The parser is designed to parse SOFL formal specifications and store syntax tree information in an xml file for reuse. In the xml file, each tag corresponds to one grammar node of SOFL. The processor will take the syntax tree information (xml file) as input, generate functional scenario forms, and store these functional scenario forms back into the xml file. We have defined the format of functional scenario form in the xml file, based on the definition of functional scenarios as we talked in section 2. Fig. 1 shows an overview constitution of the tool.

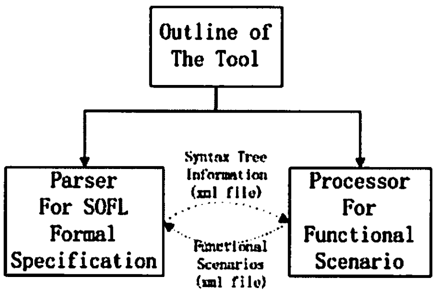

Outline of The Tool
Parser For SOFL Formal Specification
Syntax Tree Information (xml file)
Functional Scenarios (xml file)
Processor For Functional Scenario

Fig. 1. This figure shows the two core components of the tool. The parser produces the syntax tree information and creates an xml file to store it. After this, the processor will run the algorithm for generating functional scenarios, and store functional scenario forms back into the xml file.

A. Parser for SOFL formal specification

When we are to develop a parser for a specific language, generally the first task is to know what kind of lexical symbols are legal and how they are arranged in a reasonable way, defined by the grammar. The two questions determine the design of strategy for the most important parts of the parser.

We have made a summary about all legal classifications of symbols that could be accepted by SOFL. They are listed as follows in chart 1.

CHART. 1. ALL LEGAL CLASSIFICATIONS OF SYMBOLS IN SOFL.

Key value	1	2	3	4
Classification	Enumeration	Character	String	Number
Key value	5	6	7	8
Classification	Identifier	Key word	Comment	Separator

We pick the "identifier" to explain that identifiers in SOFL should maintain what kind of features and how we construct the acceptor to receive an identifier from the texture stream. The "identifier" should start with an English letter and after that could consist of letters and digits (from 0 to 9). This rule can be described in an alternative way, which is more intuitive. We build a finite state machine for accepting "identifier" in SOFL as figured in fig. 2.

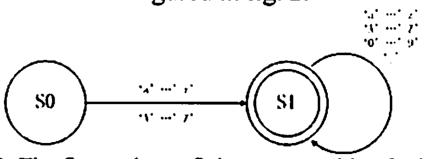


Fig. 2. The figure shows finite state machine for identifier.

For the grammar of SOFL, also could referred to Liu's publication, we find that it is defined by top-down architecture. In the most top level, the formal specification is abstracted in one grammar node, which is called "specification". The "specification" is decomposed to some grammar nodes in the lower level, which are restricted to list in order. Continuously, all grammar nodes could be decomposed further and at last reach the terminal node, which are symbols that could be accepted directly by lexical analyzer. We pick up one part of the grammar to give a straightforward image.

One part of the grammar defining the module of SOFL:

```
Module ::=
        "module" Identifier [ "/" ( Identifier | "SYSTEM"
        Identifier)] ";"
                Module_body
        "end_module"
Module_body ::=
        ["const" Const_declaration ";" ]
        ["type" Type_declaration ";" ]
        ["var" Var_declaration ";" ]
        ["inv" Inv_definition ";" ]
        ["behav" Behavior ";" ]
        Process_function_specifications
```

Here we use some notations, in which double quotation marks quote a terminal node (a symbol), brackets mean the content is optional, round brackets and vertical bars cooperate to imply a multiple selection. We can find that this kind of description is easy to understand for readers, but not suitable for syntactic analysis program. So the first work for us is to rewrite the grammar to make it adaptable for realization. The translation is like the following.

```
Module -> "module" Identifier S1 ";" Module_body
        "end_module"
S1 -> epsilon | "/" S2
S2 -> Identifier | "SYSTEM_" Identifier
Module_body -> S3 S4 S5 S6 S7
                Process_function_specifications
S3 -> epsilon | "const" Const_declaration ";"
S4 -> epsilon | "type" Type_declaration ";"
S5 -> epsilon | "var" Var_declaration ";"
S6 -> epsilon | "inv" Inv_definition ";"
S7 -> epsilon | "behav" Behavior ";"
```

Notations used here is almost the same with the ahead. Each line of these sentences is called "grammar deduction formula". In each formula, it contains only nonterminal grammar nodes and terminal nodes. Terminal nodes are symbols which could be accepted by lexical analyzer and nonterminal nodes are able to be coded as methods which are responsible for the syntactic analysis of according units.

So far, we have got an overview about the question domain of the parser. The next step is to design for the implementation. If we view the parser as a software project, we need to decompose the whole task and depict the architecture of the software. Fig. 3 shows the architecture of the parser.

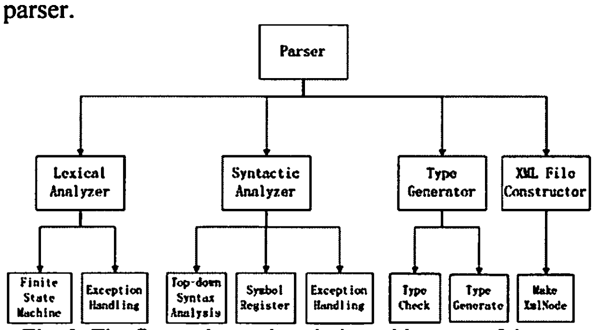


Fig. 3. The figure shows the whole architecture of the parser.

The parser includes four components, lexical analyzer, syntactic analyzer, type generator, and xml file constructor. Each component could be decomposed to the third level, of which these are able to be implemented by program units. The core function of a parser is syntactic analysis. As we explained before, the feature of SOFL's grammar determines that top-down analysis strategy is convenient. During the approach of syntactic analysis, we also need to register all symbols and make a symbol table for potential semantic analysis to some extent. Beside of these, we need to design a robust mechanism for exception handling and recovery. We use Fig. 4 to explain the main procedure of top-down syntax analysis.

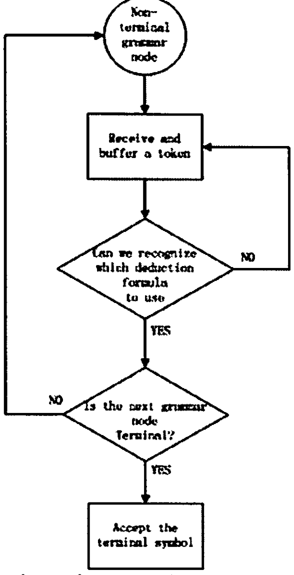

Non-terminal grammar node
Receive and buffer a token
Can we recognize which deduction formula to use
NO
YES
Is the next grammar node Terminal?
NO
YES
Accept the terminal symbol

Fig. 4. The figure shows how top-down syntactic analysis works.

Because terminal nodes could be directly provided by lexical analyzer, terminal grammar nodes usually are easy to deal with, and do not need to make methods for them. In Fig. 5, we focus on how to deal with nonterminal grammar nodes. As we mentioned previously, nonterminal grammar nodes should be realize as methods, according to the grammar deduction formulae. So when we are faced with multiple selections about which formula to use and cannot recognize it based on the left-most symbol, we choose to look forward more symbols for help.

B. Processor for generating functional scenarios

After the work of parser, an xml file is created for the storage of syntax tree information. Each section, in the formal specifications, could be extracted separately from this file. As we have analyzed before, functional scenarios are corresponding directly to the section of process in one module. At this stage, we are able to visit all well-classified information that may help for generating functional scenarios.

The processor will take this kind of xml file as input and generate functional scenarios for each process. At last, we shall organize the functional scenarios into functional scenario forms (FSF), and append FSF to its' appropriate position in the xml file. In this way, we can get access to both information of each process and its correlated FSF.

So far we have been clear about the input and output of this processor. Considering about the algorithm which has been explained in section 2, the first task should be transforming post-condition into disjunctive normal form (DNF). From now on, we are willing to take a predicate $((a \vee b) \wedge c) \Leftrightarrow (d \Rightarrow e)$ as example to demonstrate how we implement the algorithm. The predicate stands for post-condition of a process, in which a, b, c, d and e are atomic predicates. It may seem simple, but we think it is fine and enough to represents general situations when we face with this matter.

First of all, we shall build a syntax tree from what is stored in xml file. In this case, the syntax tree should look like Fig. 5.

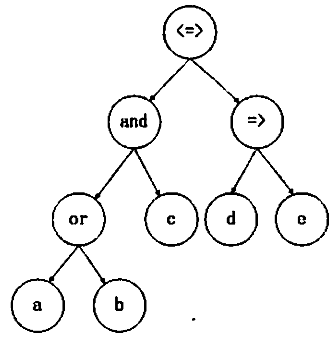


Fig. 5. The syntax tree of the example post-condition.

The algorithm for transforming post-condition to DNF is arranged as following:

Step 1: Search nodes of the syntax tree in root-first order, and for each node if the value of currently visited node is "<=>", replace the node with sub-tree like following

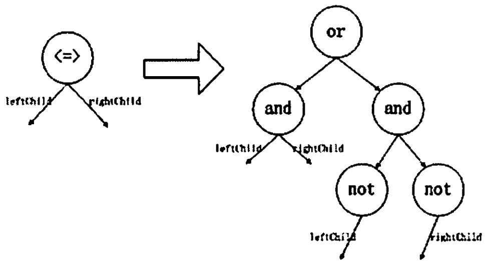


Fig. 6. Replace all "<=>" nodes in the syntax tree.

Step 2: Search nodes of the syntax tree in root-first order, and for each node if the value of current node is "=>", replace the node with sub-tree like the following

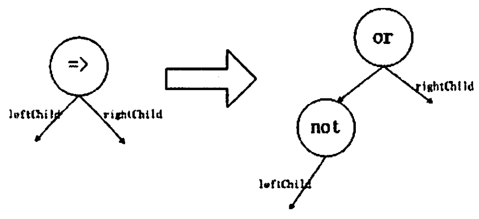


Fig. 7. Replace all "=>" nodes in the syntax tree.

Step 3: Search nodes of the syntax tree in root-first order, and for each node if there exists "not" disorder, that means "not" node's child is not atomic predicate, replace the node with sub-tree like the following

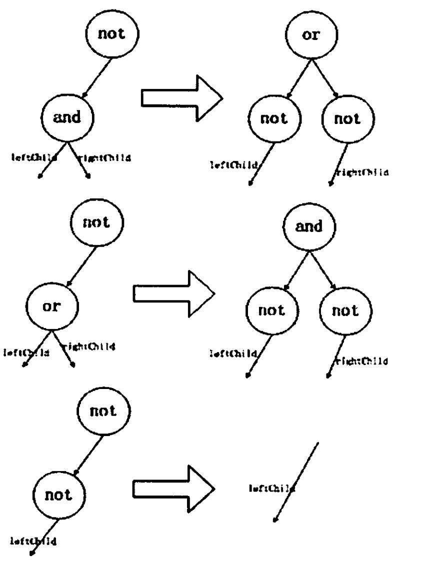


Fig. 8. The figure shows how to deal with "not" disorder.

Step 4: Search nodes of the syntax tree in root-first order, and for each node if there exists "and/or" disorder, that means "or" node becomes the child of "and", replace the node with sub-tree like the following

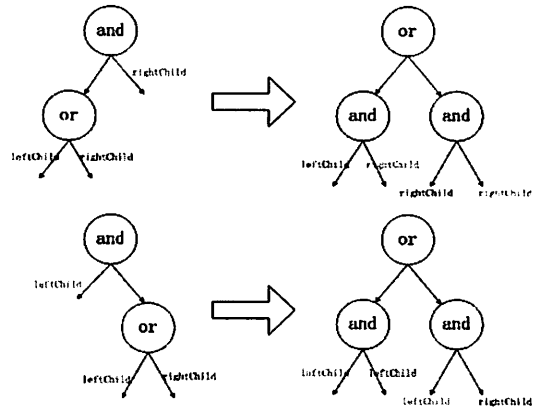


Fig. 9. The figure shows how to deal with "and/or" disorder.

Since the post-condition has been transformed into DNF, considering of the data structure we used, we have got a binary tree, in which "or" nodes are ancestors of "and" nodes and "not" nodes should be connected directly with atomic predicate nodes.

In case of the example we mentioned before, the following figure give a view of the final syntax tree, which has been transformed into disjunctive normal form. It contains no "<=>" nodes and "=>" nodes. In addition, it involves no "not" disorder and "and/or" disorder. For "not" nodes, they are connected with atomic predicate directly. And in the following figure, "not" nodes are combined to atomic predicates.

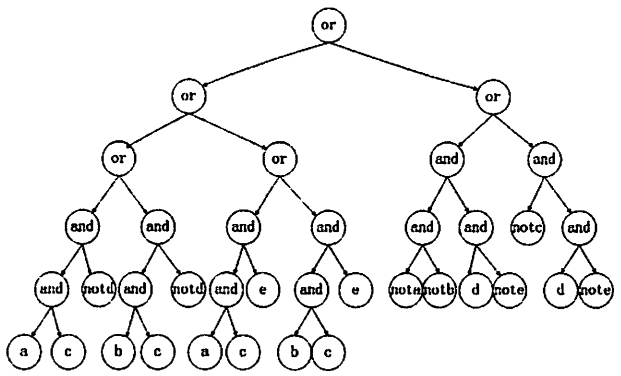


Fig. 10. The figure shows DNF of the example post-condition.

We can easily find out each conjunction clause, which is a sub-tree whose root is the first "and" node. There exist six conjunction clauses in the DNF. They are $a \wedge c \wedge \neg d$, $b \wedge c \wedge \neg d$, $a \wedge c \wedge e$, $b \wedge c \wedge e$, $\neg a \wedge \neg b \wedge d \wedge \neg e$, and $\neg c \wedge d \wedge \neg e$.

According to the algorithm of generating functional scenarios, for each conjunction clause, we need to make a partition for separating atomic predicates by whether they contain output variables. If "a" and "c" contain no output variable, they are called "guard condition". On the other side, "b", "d" and "e" are called "defining condition". The next step we shall do is to rearrange these conjunction clauses. The six conjunction clauses should be rewrite to be $a \wedge c \wedge \neg d$, $c \wedge b \wedge \neg d$, $a \wedge c \wedge e$, $c \wedge b \wedge e$, $\neg a \wedge \neg b \wedge d \wedge \neg e$, and $\neg c \wedge d \wedge \neg e$.

The last step is to combine conjunction clauses of which the defining conditions are the same. Because guard condition and defining condition of a conjunction clause are stored in an array list, by searching each node in this array list, we can judge whether one defining condition is covering the other and at the same time the other is also covering the original one. If $A \subseteq B \wedge B \subseteq A$, we can make a conclusion two set of defining conditions are same. If there exist any two conjunction clauses whose defining conditions is the same, we shall combine them by merging the guard conditions of them by connector of logic "or". In case of this example, the conjunction clauses should keep themselves, because of the defining conditions of each one are unique among them.

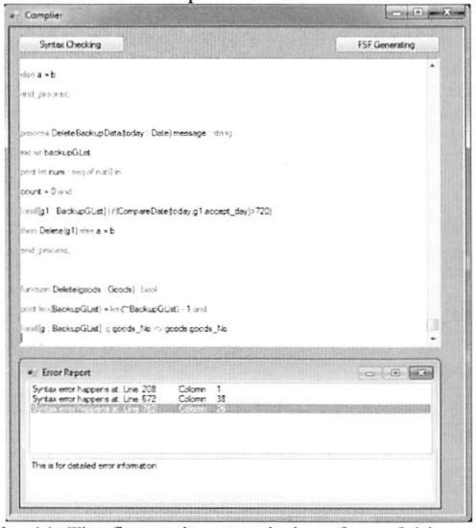
So we can finally generate functional scenarios in this form $\sim OP_{pre} \wedge C_i \wedge D_i$. All guard conditions and defining conditions of the example are listed in the following chart.

CHART. 3. GUARD CONDITION AND DEFINING CONDITION OF EACH FUNCTIONAL SCENARIO FOR THE EXAMPLE.

	C (guard condition)	D (defining condition)
Functional scenario 1	$a \wedge c$	$\neg d$
Functional scenario 2	c	$b \wedge \neg d$
Functional scenario 3	$a \wedge c$	e
Functional scenario 4	c	$b \wedge e$
Functional scenario 5	$\neg a$	$\neg b \wedge d \wedge \neg e$
Functional scenario 6	$\neg c$	$d \wedge \neg e$

C. Interface of our tool

We tested this tool by several SOFL formal specifications, one of which defines the requirements of a goods delivery system. We take that example to show interface of our tool.

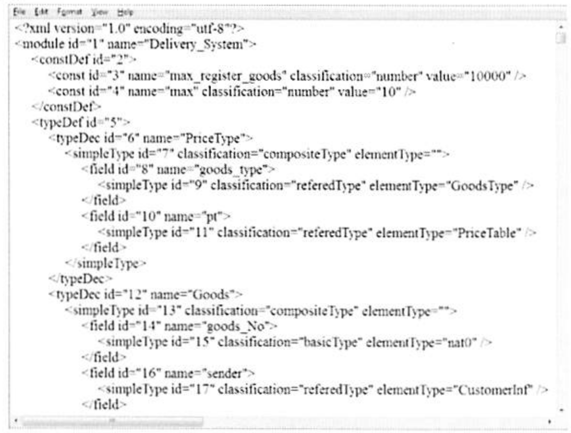


```
else a = b
end_process;

process DeleteBackupData(today : Date) message : string
ext wr backupGList
post let num : seq of nat0 in
count = 0 and
forall[g1 : BackupGList] | if(CompareDate(today,g1.accept_day)>720)
then Delete(g1) else a = b
end_process;

function Delete(goods : Goods) : bool
post len(BackupGList) = len(~BackupGList) - 1 and
forall[g : BackupGList] | g.goods_No <> goods.goods_No
]
```

```
Syntax error happens at  Line 208     Colomn   1
Syntax error happens at  Line 672     Colomn   38
[illegible]
This is for detailed error information
```

Fig. 11. The figure shows main interface of this tool.

The central textural area demonstrates the SOFL formal specification. If there exist some syntactic errors, the tool will collect information for the position where error happens and report them in the below frame. Key words are highlighted in blue and the character where error is located is highlighted in red. After parsing, an xml file storing both syntax tree information and functional scenario forms is created. For this test case, corresponding xml file is showed in the following figure.



```
<?xml version="1.0" encoding="utf-8"?>
<module id="1" name="Delivery_System">
  <constDef id="2">
    <const id="3" name="max_register_goods" classification="number" value="10000" />
    <const id="4" name="max" classification="number" value="10" />
  </constDef>
  <typeDef id="5">
    <typeDec id="6" name="PriceType">
      <simpleType id="7" classification="compositeType" elementType="">
        <field id="8" name="goods_type">
          <simpleType id="9" classification="referedType" elementType="GoodsType" />
        </field>
        <field id="10" name="pt">
          <simpleType id="11" classification="referedType" elementType="PriceTable" />
        </field>
      </simpleType>
    </typeDec>
    <typeDec id="12" name="Goods">
      <simpleType id="13" classification="compositeType" elementType="">
        <field id="14" name="goods_No">
          <simpleType id="15" classification="basicType" elementType="nat0" />
        </field>
        <field id="16" name="sender">
          <simpleType id="17" classification="referedType" elementType="CustomerInf" />
        </field>
```

Fig. 12. The figure shows the main output of this tool: xml file storing syntax information and functional scenarios.

IV. RELATED WORK

Tool support is significantly important for applying formal methods into practice. There are many formal languages with powerful tool. One famous example is VDM. As introduced in [6], "Overture" is developed to be a common open-source platform integrating a range of tools for constructing and analyzing formal models of systems using VDM. Nowadays, "Overture" has been updated to be a stable and mature platform, but restricted by features of VDM, it could not support effectively on describing the architecture of whole software. There are also other tools supporting formal language like JML and Alloy. They are introduced in [7][8].

Functional scenario-based techniques also gain more and more attentions in research. [1] proposes an automated functional scenario-based formal specification animation method. [2] talks about their work on an experiment for assessment of a functional scenario-based test case generation method, which is improved from FSBT (functional scenario-based testing) proposed in [9]. For functional scenarios' generation, a method for automatic generation of functional scenarios from SOFL CDFD has been talked in [10].

ACKNOWLEDGMENT

I want to express my acknowledgement to professor Liu and all partners in Liu's lab. Professor Liu made a solid foundation of SOFL for me to understand, and during designing the functions of this tool, he gave many advices that makes this tool easy to use. My acknowledgement also goes to all partners in Liu's lab. They helped me both on technique issues and how to organize this paper.

REFERENCES

[1] Mo Li, Shaoying Liu, "Automated Functional Scenarios-based Formal Specification Animation", 19th Asia-Pacific Software Engineering Conference (APSEC 2012), IEEE CS press, Hong Kong, 2012 (to appear).J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68-73.

[2] CenCen Li, Shaoying Liu, Shin Nakajima, "An Experiment for Assessment of a "Functional Scenario-based" Test Case Generation Method", International Conference on Software Engineering and Technology (ICSET 2012), 2012, pp. 64-71.K. Elissa, "Title of paper if known," unpublished.

[3] Shaoying Liu, A Jeff. Offutt, Chris Ho-Stuart, Yong Sun, Mitsuru Ohba, "SOFL: A Formal Engineering Methodology for Industrial Applications", IEEE Transactions on Software Engineering, Vol. 24, No. 1, January 1998, pp. 24-45.

[4] Shaoying Liu and Shin Nakajima,"A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications", 4th IEEE International Conference on Secure Software Integration and Reliability Improvement, Singapore, June 9-11, 2010, pp. 147-155

[5] Shaoying Liu, Yuting Chen, Fumiko Nagoya, John McDermid, "Formal Specification-Based Inspection for Verification of Programs", IEEE Transactions on Software Engineering, 38(5), 2012, pp.1100-1122.

[6] Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative Integrating tools for VDM. ACM Software Engineering Notes 35(1) (January 2010).

[7] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. Draft, available form jmlspecs.org, 2005.

[8] Daniel Jackson. Alloy: a lightweight object modeling notation. ACM Trans. Softw. Eng. Methodol., 11(2): 256-290, 2002.

[9] Shaoying Liu and Shin Nakajima,"A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications", 4th IEEE International Conference on Secure Software Integration and Reliability Improvement, Singapore, June 9-11, 2010, pp. 147-155.

[10] Mo Li, Shaoying Liu, "Automatically Generating Functional Scenarios from SOFL CDFD for Specification Inspection", 10th IASTED International. Conference on Software Engineering, Innsbruck, Austria, Feb. 15-17, 2011, pp. 18-25