International Journal of Computing Technology and Information Security
Vol.1, No.1, pp.88-98, March, 2011

Soflipse: Tool for Automatic Modelling and Reviewing SOFL Workflows

Yisheng Wang, Qing Zheng and Haopeng Chen

School of Software, Shanghai Jiao Tong University
Shanghai, 200240, China
easonyq@hotmail.com, mark.q.zheng@gmail.com, chen-hp@sjtu.edu.cn

Abstract: *Software quality is always a hotspot discussed by developers around the world. Formal method in developing software is a new topic that attracts many researches' attentions. It uses formal language to describe and model key workflow in a software project and thus uses mathematical and logical inference to ensure the correctness of the workflow. SOFL (Structured Object-oriented Formal Language) is a kind of formal language describing the input, output and processing procedure of a service accurately. There are two parts in SOFL, SOFL Specification and CDFD Diagram, which describing a workflow from two different angles. Besides, some basic elements are introduced into SOFL such as Module, Process, Dataflow, etc. We can also review and validate a workflow modeled by SOFL through strict formal methods. In this way we can finally build up a formal workflow without any syntax or obvious errors. In order to show the feasibility of using SOFL to finish the software developing job, we have also developed plug-in tool called 'Soflipse' which enables the users to model and review a workflow automatically. By using this, the users can ensure that any workflow which passes through review and validate test won't crash and will get expected answers comparing with its requirements.*

Keywords: *Formal method, SOFL, Softlipse, Workflows;*

1. Introduction

A commercial software product always needs tens of months' time and the cooperation of many people, including coders, testers, and designers. Nevertheless, software quality still can't be guaranteed. Bugs, maintaining and costs are the bottleneck of software industry [11]. We can never prove that a software product is correct, for its infinite input can't be enumerated [6]. Every software product has its potential problem and no one can tell whether it will crash in the next second. What's more, in distributed system where a single application mostly need the collaboration of several hundred computers, software errors occur at a fairly high rate. People have already found a lot of ways to improve software quality such as standard developing processes and complete software testing. Using these classic methods such as RUP developing process, black-box testing or white-box testing can indeed improve its quality, but still 100% correct is unable to be reached or proved.

Actually, there are some other studies that focus on formal methods in software developing. SOFL [14] is a kind of formal language which can be used to describe a workflow. Usually we use SOFL Specification and CDFD to model a workflow. There also exist some basic elements in SOFL, such as Module, Process, Dataflow and so on. CDFD describe mostly the relationship between these basic elements, such as what kind of dataflow exists between two different and neighbored processes, or the overall dataflow of the whole process and module. Moreover, there also exists hierarchical relationship between two CDFD Diagrams. This is similar to Class Diagram in UML [16].

In SOFL, if a process can be decomposed into a module with several processes in children-level, its child-level CDFD Diagram describing the module it decomposed into can also exists as the refinement of the parent-level CDFD Diagram. The input and output dataflow in child-level CDFD is the same as them of the process in parent-level. Compared with CDFD Diagram, SOFL Specification describes mostly the detailed information of every basic element,

such as how the detailed implementation of a process is, or what kind of internal and external variables or constants a module has and so on. More vividly, we can make a comparison with Entity-Relationship Diagram and SQL script in database designing [2]. In database designing, we also have two kinds of describing method, a visual one and a textual one. So CDFD Diagram is similar to Entity-Relationship Diagram, which describes relationship between basic elements, and SOFL Specification is similar to SQL script, which describes the detailed method. But different with database designing, Entity-Relationship Diagram can be transformed to SQL script but CDFD can't. CDFD Diagram and SOFL Specification work has complementary roles. We can't describe and model a software process with only one method of them, otherwise we would lose some information of software, and the principle of equivalent is violated. Only by using these two methods at the same time can we describe a software process in a complete and accurate way.

Formal method is more complicated compared to classic methods such as UML. But its ability of review and validate is the key advantage over UML [24] [13] [12]. So by analyzing SOFL Specification and CDFD, we can find whether there are some potential errors in the model by formal methods which will be described afterward. And we should keep in mind that workflow describing with SOFL can be ensured to be correct. Here we want to point out the difference of review and validate towards a formal language. Review means checking the textual specification or codes and trying to ensure it can run correctly without any syntax or static semantic errors like constant value violation. Validate means checking whether the written formal modeling fits the need proposed at begin. These are two different steps formal methods use to ensure the correctness of codes.

We have developed a plug-in tool namely ‘Soflipse' based on IBM Eclipse platform [1]. This tool helps the users to model and review a workflow automatically. Through Soflipse, users can describe and model there software process by drawing CDFD Diagram. After that system will generate the SOFL Specification according to the CDFD and then begin to review and validate it. If there are some errors in SOFL Specification, the editor will show it by make the error codes red-underlined. Otherwise, it will show a success information which means it passes reviewing.

The rest of the paper is organized as follows: Section 2 lists some motivations of our work. Section 3 introduces some architecture and implementation of our tool. Section 4 shows our approach about how to review SOFL. Section 5 summarizes some experiment evaluation of the tool including a sample workflow and its result. Section 6 introduces related work by others recently. Section 7 summarizes the main contribution of the paper and comments on further research.

2. Motivation of the Research

Software today is widely used in our daily life. Among them, some applications like Bank System, Military System, etc ask for a fairly high quality need, for once if there is a mistake; the loss would be great and couldn't be made up usually [10]. For these reasons, we have to improve the quality (including availability, usability, repair time, etc) of these important software product. There's a study named ‘Software Engineering' which focuses on how to develop a high quality software product by using standard developing process such as RUP or XP. By using these, software quality can indeed gain improvement, but never reach perfect level. So we can always see these sentences in Software Requirement Documents: Our product can run correctly within 99.99% of the time. Theoretically this correctness rate can be very close to 100%, but never reached. That's the reason why even today we often see the news that some bank suffered a great loss because of the incorrect action its ATM did.

Formal method can help us to solve this problem. Researches on formal method used in software development begin in the 70’s last century. It is Dijkstra's Weakest Pre-predicate Calculus and Hoare Logic that started the research and proved it feasible theoretically. Unlike classic methods, a workflow modeled by formal language can be proved correct using mathematical and logical method. After several steps of inference, we can judge whether the

piece of formal language codes fits its need. If so, we can say that this piece of codes is correct, which means we can run it without most of errors. (More information will be discussed afterward.) And this inference process is called review.

Only using formal language to describe and review a software workflow isn't enough. What we seek is to review it automatically, rather than we humans use our brains to do it. So we should set up a standard format which enables computer to know what formal language is talking about and how to review it. In this paper, we use SOFL as our language and we have also developed a tool which lets users to build up his own SOFL workflow. After that, he can review it just by clicking a single button. The entire review job will be done after this click automatically. Review can help us find some obvious errors such as parameter type mismatch or constant value violation. We can see it in the following example.

Suppose we have such a workflow composed by two processes. The first one returns an integer x as its output with the post-condition x > 5. The second one takes an integer y as its input with the pre-condition y > 6. We can clearly see that Process 1 can't meet the pre-condition of Process 2, which we call it constant value violation. If we change the input variable of Process 2 into character y, which is called parameter type mismatch.

It must be noted that in this paper we want to point out an automatic model and review method. Comparing with those half-automatic methods, no humans are involved in reviewing the workflow, which means all the mathematical and logic inference are done by computer and algorithm.

As conclusion, our goal is to find all these kind of errors by using the proposed tool and to ensure the workflow is correct. And also to ensure that the whole process is automatic using 'Softlipse'.

3. System Implementation

As an eclipse plug-in project, our tool has a similar architecture like other eclipse plug-ins. When using it, all the user need to do is to unzip the RAR file into eclipse's installing directory. We divided our system into three layers, which can be seen as Figure 1.

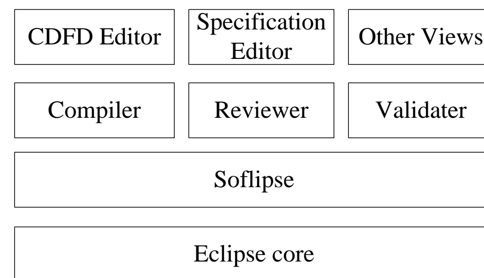

Figure 1. Architecture Diagram

Presentation Layer

The first layer composed by 'CDFD Editor' and 'Specification Editor' acts as the presentation layer. It provides users two editor for different usage. CDFD Editor is the default editor of the SOFL modeling file (with suffix '.module'). In this editor, user can draw his own CDFD by using tool bar beside the editor. User can drag and drop processes use link line to connect them and fill in the pre and post condition to complete a process. All of these jobs can be down with the cooperation of CDFD Editor and other views such as the outline view, properties view, etc. This graphical editor extends GEF (stands for Graphical Editor Framework) which allows developer to build up his own editor with different style. After drawing and filling in information, user can switch to Specification Editor to see the auto-generated SOFL Specification and any syntax errors if exist. This textual editor is set to read-only, in order to prevent user modifying it casually and violating SOFL's rule. We have also done a red-underline mark to show any types of errors if exist. These errors can be checked by Compiler, Review or Validate in the lower layer, which will be discussed in the next paragraph. What's more, for better user experience, we also overwrite some common views which will be used frequently. These views including Package Explorer (shows resource tree and its inner structure), Outline (shows the hierarchy relationship within the CDFD) and Property (shows detail information and allow user to fill in some of them) These views help user to complete his SOFL model, such as filling pre and post condition using Property view.

Logic Layer

Below the presentation layer, we build up a logic layer which is responsible for some jobs involving inner logic of SOFL. Here are three modules. The compiler takes SOFL Specification codes as input, and returns a semantic tree and all syntax errors if exist. We use JavaCC as our tool to write this SOFL Compiler and all the SOFL grammar is lists behind books written by Shaoying Liu, who at first proposed and made research on SOFL. Some detail but important problem like left-recursive, priority of operator can be found in Andrew W. Appel's books [3]. If there are any syntax errors such as semicolon missing, undefined identifier or parameter number mismatching, the compiler will check it out and report to the Specification Editor, and finally show user in the form of red-underline. Review and Validate is the key functional module which reflects the value of formal methods. Reviewer takes semantic tree as input which generated by compiler and judge whether the semantic tree fits SOFL's rule. In this process, reviewer also needs some link information provided by CDFD Editor, for this type of information isn't shown in SOFL Specification. After review process, a result will be generated to point out whether this workflow can pass review or any possible errors. The last module named ‘Validater' ensures the consistency between SOFL model and customers' need. Because this paper we focus on SOFL modeling and review job, so validate is out of range. For more information about validate, readers can refer to Qing Zheng's paper about validate in SOFL.

Plug-in Layer

The third layer named ‘Soflipse' is just a virtual plug-in layer connecting to Eclipse Core. As a eclipse plug-in project, we must register itself to eclipse for its recognition and communication. This layer is responsible for the information exchange between our system and eclipse. For example, when user tries to rename a process using Properties View, it will send a message to eclipse core and then a rename event will be fired to notice CDFD Editor changing its display name. It acts as a connector and information collector for our system.

4. Approach

How to realize the function of compiling and reviewing SOFL will be discussed in the following. In general, the SOFL Compiler takes SOFL Specification codes as input and returns a semantic tree to show its inner hierarchy relation. In this process, any syntax or semantic errors will be thrown. Review in SOFL means checking SOFL Specification to ensure that it can run without errors which are possible to lead it to crash. The reviewer takes semantic tree and CDFD as input and then checks whether there exist errors. After checking errors including syntax, semantic, or higher level such as neighbor processes mismatch, we can say that a workflow without these is correct.

Compiler: We use JavaCC as our compiler tool. JavaCC is a compiler generator which needs user to write a profile with suffix ‘.jj' and generate a compiler written in Java. All the lexical unit and syntax rules are written in the jj file. More specific, JavaCC use LL(1) as its compile algorithm which means from left to right, left-most derivation with looking ahead 1 word each time. More information about JavaCC and LL(1) you can refer to some books about compiler.

Actually, using JavaCC as compiler generator isn't as easy as I described above. If we directly input all the SOFL syntax into jj file, we will surely run into these two problems: left-recursive and operator priority. I'll introduce these in the following.

Left-recursive: Left-recursive means a compiler using LL(1) algorithm come into infinite loop because of the grammatical problem. The result will be stack overflow and the algorithm can't reach end. Let's consider a grammar rule as Figure 2.

$$S \rightarrow \exp$$
$$\exp \rightarrow \exp + \exp$$
$$\exp \rightarrow 1,2,...,9$$

Figure 2. Left-recursive Grammar

Obviously this is a grammar describing operation plus. But when compiler tried to match this grammar, it found that ‘exp' can

always expand to 'exp + exp'. That simply comes to a infinite loop. So this is the problem of grammar, not compiler. What we should do is to change the grammar into non-left-recursive one by changing its structure while keep it equivalent to the original one. Figure 3 is the possible one without left-recursive.

$$
\begin{aligned}
&S \rightarrow \exp \\
&\exp \rightarrow \exp_1[\exp_1{}'] \\
&\exp_1 \rightarrow 1,2,...,9 \\
&\exp_1{}' \rightarrow +\exp_1[\exp_1{}']
\end{aligned}
$$

Figure 3. Non-left-recursive Grammar

With the introduction of exp1', the grammar is able to stop in several step. The key to this problem is that in the last grammar First (exp) = exp while in this one First (exp1') = +, where '+' is a terminal character. We need to transfer all these left-recursive grammars into non-left-recursive ones to ensure the compiler generated won't come into infinite loop.

Operator Priority: We know in mathematics, any binary operators has its priority, such as multi and divide is prior to plus and minus. Here as a compiler we also need to deal with these things appearing in expression. Unlike elementary mathematics where only four types of basic operator are defined, SOFL has defined 20 types operator. We divided them into 8 levels, which are listed from low to high in terms of their priority.

1. <=>
2. =>
3. or
4. and
5. =, <>, <, <=, >, >=, inset, notin
6. +, -
7. *, /, div, rem, mod
8. **

As we have discussed above, binary operator can lead to left-recursive problem. So according to last section, you may change the original grammar contains 'exp → exp binaryop exp' into ones as Figure 4.

$$
\begin{aligned}
&S \rightarrow \exp \\
&\exp \rightarrow \exp_1[\exp_1{}'] \\
&\exp_1 \rightarrow 1,2,...,9 \\
&\exp_1{}' \rightarrow \mathit{binaryop} \;\; \exp_1[\exp_1{}']
\end{aligned}
$$

Figure 3. Binary operator grammer without left-recursive

Still we haven't made operator priority into consideration. It can be easily seen that according to this grammar describing in Figure 4, any operator appears earlier in an expression will get higher priority. So such an expression like '3+5*2' will get the answer 16, rather than 13. Thus we need to improve further in base of the grammar shown in figure 5 to make it able to judge priority.

$$
\begin{aligned}
&S \rightarrow \exp \\
&\exp \rightarrow \exp_1[\exp_1{}'] \\
&\exp_1 \rightarrow \exp_2[\exp_2{}'] \\
&\exp_1{}' \rightarrow lv1op \;\; \exp_1[\exp_1{}'] \\
&\exp_2 \rightarrow \exp_3[\exp_3{}'] \\
&\exp_2{}' \rightarrow lv2op \;\; \exp_2[\exp_2{}'] \\
&...... \\
&\exp_8 \rightarrow 1,2,...,9 \\
&\exp_8{}' \rightarrow lv8op \;\; \exp_8[\exp_8{}']
\end{aligned}
$$

Figure 4. Binary operator grammer

Consider grammar shown in Figure 5 where lvxop means operator with x as its priority level. Let's imagine the compile process according these grammar with the expression '3+5*2'. In this expression '*' of level 7 and '+' of level 6 appears. So when the compiler runs into '+', it will build up a tree with parent node '+', left child node '3' and blank right child node. After that, when it comes across '*' with higher level, it replace the blank right child node with '*' and its two child '5' and '2'. And if we change this expression into '3*5+2', things are different. After it meets '*' and builds up a tree with parent node '*', left child node '3' and blank right child node, it won't replace the blank right child node with '+', for '+' gains lower level than

'*', and once the compiler trap into higher level operator, it won't return back to construct lower ones. This time, it will replace the blank right child node with '5', and build a new tree with parent node '+', left child node '*' and right child node '2'. At last, we expand the grammar from 3 lines to 18, but we solve the operator priority and left-recursive problem, which is very important for a compiler developing.

Reviewer: The notion of the word 'review' varies from many researches. In SOFL, the aim of review is to ensure that the workflow modeled by formal method can run well and without most of errors, which has been indicated earlier. Researches on finding methods to review workflow modeled by formal language has attracted great attention and people indeed contribute a lot in this field. Shaoying Liu has proposed a review method called 'review task tree', which is similar to fault tree analysis. What should be pointed out is that In Liu's method, he doesn't build up a SOFL Compiler, which helps to check lexical, syntax and semantic errors in SOFL. So review task tree is responsible for all errors that should be checked in review, which ranges from lexical to high level. That is the main difference between our method and review task tree.

The coverage of errors which can be detected by reviewer is limited. Some variables that need user to give can't be checked until they are actually assigned. For the reason that user's input can't be forecasted, these variables can't be reviewed before run-time. In other words, only static variables or expression can be reviewed. Moreover, in our system SOFL Specification is auto-generated and read-only, which means users are not allowed to modify it casually. This helps us to lower the complexity of checking errors, as a result, reviewer in Soflipse only needs to check the matching of parameters from pre and post condition of neighboring processes. Any errors like syntax or semantic has already been found out in SOFL Compiler.

Another point that readers should keep in mind is that only contradiction expression will be checked out, for there doesn't exist a single value that satisfied this expression. For any expression, if there exist one (or more) value(s) satisfying the expression, it shouldn't be checked out as errors. For instance, for any integer x, expression x>5 and x<5 is a contradiction expression which should be checked out, while another expression x>5 and x <=5 is not.

Reviewing job can be divided into two steps, which will be discussed in the following:

Existence Review: Existence review is the first step of reviewing which checks correctness of parameter of a process. Correctness here includes parameter number, sequence, etc. Formally, let's define,

- $N_i(P)$ as the number of input parameter of process P;
- $N_o(P)$ as the number of output parameter of process P;
- $input(P)$ as the input stream of process P;
- $output(P)$ as the output stream of process P;
- $I_i(P,v)$ as the index of the input variable v in the input stream;
- $I_o(P,v)$ as the index of the output variable v in the output stream;

and we assume that P1 and P2 are two neighboring processes, they must satisfy:

1. $N_o(P_1) = N_i(P_2)$
2. $forall[v : output(P_1)] \mid v \ inset \ input(P_2)$
3. $forall[v : output(P_1)] \mid I_o(P_1, v) = I_i(P_2, v)$

The first equation proposed that if two processes are neighboring, then the number of parameters of these two processes must be equal. The second formula means that if two processes are neighboring, for all the parameter variable appears in the first process must also appear in the second process. In other words, it was not allowed that the second process used some variables that didn't appear in the first process. The third formula indicates that the output stream of the first process must be the same as the input stream of the second process, including their indexes and sequences.

It can be inferred from these three rules that any errors such as parameter number difference or type mismatching are not allowed. It should also be noted that parameter types are not necessarily be the same. In SOFL, all types are structured like a tree just as JAVA. For example, integer can be cast to double, and nat can also be cast to nat0 which composed by nat and 0. One type can always be cast to another type whose range is bigger than the original one, but not vice versa.

All these parameter information can be gotten from SOFL Specification. And all the link information among processes is provided from CDFD. Checking these three rules won't take a long time, so it is able to be implemented and used.

Satisfying Review: Those processes which pass existence review can't be treated as correct ones, for they also have to pass the second step, satisfying review. Satisfying review means two processes are neighboring but the post-condition of the first process will never satisfy the pre-condition of the second process. It is clearly that pre-condition and post-condition always return boolean values. So applying the knowledge from discrete mathematics here, these two conditions can be treated as two propositions (named P and Q), and we need to judge the value of the expression $P \rightarrow Q$ [19]. The value of this expression totally has three situations: tautology, contradiction and uncertain situation. Just as what has already been pointed out, only contradiction should be reviewed, for at least one value satisfied the expression in tautology and uncertain situation [7]. If $P \rightarrow Q$ is contradiction, this workflow may be not able to run to the end. So users should be noticed about these types of errors.

According to the SOFL grammar, pre and post condition are both expressions, which can still be divided into six types: unary-expression, apply-expression, basic-expression, quantified-expression, negation-expression and relational-expression. Among these six types, relational-expression, quantified-expression and binary operator in basic-expression can lead to contradiction. (Actually negation-expression can also lead to contradiction, but this kind of expression is only opposite to another expression, so it can also be ignored.) These three situations will be discussed further.

Relational-expression is used to judge the type of an expression according to SOFL grammar. For instance, is_int(2+3) is a simple relational-expression. In SOFL, each variable must has its type when they are declared, so actually its type is confirmed and available, which means results of all these relational-expressions can be calculated and obtained in compile-time. Thus, whether it will lead to contradiction is also easy to be inferred.

Quantified-expression in SOFL is similar to quantifier in logic. It can also be divided into universal quantifier and existential quantifier. Both this two types are composed by declaration part and assertion part. What satisfying review needs to do is to check whether they share a same structure in assertion part. If so, according to different structure, different judge method should be introduced. Also take the previous expression as example, if there exists another universal quantified-expression whose assertion part is ‘x = temp mod 2', then we know these two share the same structure, and structure ‘mod' also tells us that these two are tautology, for 4 and 2 are not coprime. Surely, the operator ‘mod' won't lead to contradiction, but some operators like inset and notin will, and the system should be able to detect errors like these.

The word ‘structure' in the last paragraph has the following conception: Two expressions share the same structure if their parent nodes share the same type. It is obvious that universal quantified-expression won’t lead to contradiction with the simple plus operator. So in most cases, contradiction happens only when two expressions have the same structure. What's more, assertion part is an expression recursively, so the review algorithm should also be recursively invoked.

For binary operator in basic-expression, their structure also needs to be checked. But here comes a different point comparing with quantified-expression, which is that contradiction is still possible to be detected even when two operators have different

structures, for instance x < 5 and x > 6. As a matter of fact, binary operators have more close relationships among themselves. Some operators can be classified into the same operator system, such as ‘<=' and ‘<', ‘>=' and ‘>', while others express the opposite meaning, such as ‘=' and ‘<>', ‘inset' and ‘notin'. Mathematical inference can be done within same operator system and opposite operator system, like x > 5 and x >=6. Usually, contradiction happens in opposite operator system, so more emphasis should be put on these operators.

Another interesting topic is that many binary operators have a feature called transitivity, which allows us to simplify some expressions like x > y + 3, y > 5 is equivalent to x > 8. And the problem about what kind of operators can be simplified while others can't still depends on the operator system they are located in.

If a workflow passes satisfying review, it is proved that it doesn't contain contradiction, which in other words means that there at least exists one situation that it can run correctly. But it doesn't mean this workflow can run correctly on every situation no matter what the input and environment is, unless it is a tautology. This is a classic mistakes many people misunderstanding about formal methods.

5. Evaluation

We have designed a sample workflow and using Soflipse to model and review it. Our sample workflow is a small part of complicated software which can always be seen in examination score management. In university, when a semester ends, professor needs to register all his students' score one by one. Each time he succeeded in registering a score to database, the system adds the new score together with all the scores already existed in database and calculate an average score. In this short workflow, many basic elements are contained such as processes, input/output stream and database operation. In order to show as many function of SOFL modeling as possible, a sub-process is used to calculate the average score. For the reason that this sample workflow is picked-up from real-life, so it can be believed that if SOFL can model this workflow well, it is managed to model many usable and common workflow around our life.

After analyzing this workflow, we can divide it into two big steps which is registering score and calculating average score. For the first step, it takes new score and student's ID as input and returns his score as output, for it will still be used to calculate average score. Database accessing is also necessary for registering score, which in SOFL named existing data store. For the second, calculating average score has been decomposed into a sub-module, which is made up of calculating sum of all scores and calculating average score. In Soflipse, we can easily build up a CDFD meeting the requirements above. Figure 6 shows the CDFD of the top level while Figure 7 indicates the sub-CDFD of the decomposition ‘calculating average score'.

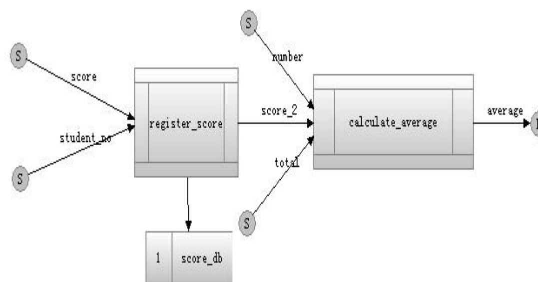

Figure 6. Top level CDFD of sample workflow

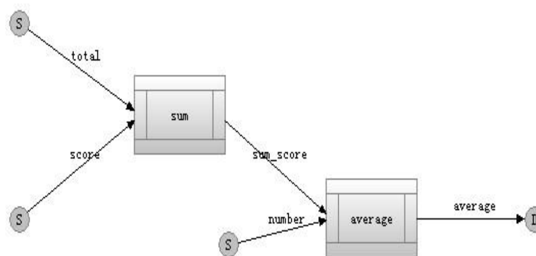

Figure 7. sub-CDFD of sample workflow

Next step is to fill in some key information to complete the model. For example we can define some type and constants, such as score_type = score, student_no_type = string, or score_max = 100 and score_min = 0. Thus, we can give out the pre-condition of the first process, registering score, which is:

score <= score_max and
score >= score_min and
student_no <> "" ……………… (1)

And its post-condition is:

```
score_db_1=override(~score_db_1,
map:{student_no -> score}) and
score_2 = score                  ……………… (2)
```

Here, the first equation of post-condition uses a built-in function named ‘override' in map, which means override the first map into the second one. And in SOFL (and many other formal language), variable with prime mark means the same variable but of old version. So this equation indicates that adding a new pair from student_no to score to the map stored in database. Pre-condition and post-condition of the second process is similar as the first one, so it will be omitted.

Besides pre and post-condition of each process, there also exists a SOFL function in the sub-module which is responsible to calculate sum of all scores.

```
function calculate_sum (score:score_type,
                 total:int):int ==score+total
end_function;
```

What should also be noticed is that variable and function in SOFL also have hierarchy relationship like many object-oriented programming languages. Variables or functions declared in the parent module are available to all its children modules, but not vice versa.

After finishing the job of completing main information in CDFD, we can switch to SOFL Specification editor to see whether there are some codes red-underlined. Moreover, a semantic tree will be displayed in the Properties view under the main editor, which shows the main structure of this SOFL model.

Through the semantic tree, we can see much information about this SOFL model, and review can also be done relying on this information. One more important thing is that SOFL modeling and reviewing job doesn't really costs so much time as we thought about formal methods before. But frankly speaking, when coming across large-scale and complicated workflow, things may have changed. Performance is always the bottleneck of the formal methods in developing software.

6. Related Work

With the rapid development of workflow technology, it is very important for workflow developer to describe the workflow specification precisely. Aiming at BPEL lack of a formal semantics and contains ambiguities, several research have been taken to formalize BPEL [15], using automata, process algebra, and Petri nets and so on.

Automata is a public and base model of formal specification for systems [9], which contains a set of states, actions, transitions between states, and an initial state, so it is convenient to describe the workflow. Diaz [4] shows a case study on converting business processes written in BPEL-WSCDL to timed automata. In paper [5], Fu develops a tool to translate the BPEL specifications to guarded automata. Although automata can well describe the BPEL, in terms of large scale system and its limitation of describing complicated functions, automata's accuracy cannot be guaranteed.

Process algebras can be also used in describing the workflow. It can be divided into many forms, such as ACP (Algebra of Communicating Processes), CCS (Calculus of Communicating Systems), CSP (Communication Sequential Process) and so on. Wong [20] discusses the workflow model described by AGP. In terms of formal verification technology, Salaün [17] presents a method of verifying business processes based on processing algebras with a particular focus on their interactions. In paper [18], Salimifard presents the translation rule between BPEL and process algebras. However, process algebras cannot support dynamic process instantiation and correlation set. It also cannot detect the dynamic structure alteration.

As Petri nets have rigorous and profound math fundamental, it can be used to analyze and verify workflow strictly. There are many researches on building workflow model based on Petri nets. It is a prevalent method on describing business process using the theory of Petri nets. Papers [8] [21] [22] describe the translation rule from BPEL to Petri nets. In paper [23], the authors can translate composition specified in BPEL into CP-nets, which can be analyzed and verified by many specialized tools. However, Petri net is still based on graphical notation and its

expressiveness is limited for large scale systems. Besides, for those system involving rich data types and high logical complexity specification, Petri nets no more can't describe precisely.

Comparing to these researches which is comparatively mature, SOFL is kind of new comer to this family. It was first proposed in the 90s of the last century and didn't attract many people's attention even till now. But as listed above, all those formal methods have a common disadvantage which usually needs humans to be involved. Using SOFL to review and validate is done automatically, whose mathematical inference can be done by computers itself. Of course, not all the workflow can be modeled and reviewed by SOFL now with the limitation of its express ability and inference performance, it is still convinced that as the researches on SOFL goes on, formal methods and SOFL will get more usage.

7. Conclusion

As the wide spread of software using in our daily life, the requirements for software with higher quality, better performance and lower cost of maintenance are become increasingly high. Using classic methods like standard developing process and complete software testing really contribute to make software better and better, but these seem not enough. Another way that helps us to improve the quality of software is called formal methods, which is trying to use mathematical and logic inference to ensure the correctness of software. The proposition of formal methods is later than many mature technologies like iterative development, but its potential attracts much people and will become a hot point for future research.

SOFL is a relatively new formal language which can be used to model, review and validate workflow from software. SOFL Specification and CDFD are both used to describe a workflow from textual and graphical angles respectively. Using SOFL to model a project is quite convenient and efficient, for it combines graphic and text together. What's more, a lot of researches have already been made on the review and validate of SOFL-modeled workflow. Our method can be generally divided into three steps: build up a compiler to eliminate lexical and syntax errors; analysis the syntax tree and transfer it to semantic tree to find out semantic errors such as undefined variable or duplicate function name; review its existence and satisfying to see whether there exists contradiction. After all these jobs, we can prove that a workflow is correct which means it won't crash or terminate by structural errors.

In order to implement and prove the feasibility of our methods, we build up an eclipse plug-in project named Soflipse. It is a tool that enables users to draw CDFD of his own workflow and help to compile, review and validate SOFL Specification which generated automatically after the finish of CDFD and other information such as pre and post-condition. We have also built up a sample workflow and test it on our system, and the result seems to be consistent with what we expected.

Next we are mean to continue doing some researches on SOFL and improve the Soflipse system. Our future work consists of three parts:

- Building up more sample workflow and test it on Soflipse to make it more robust and modify it if there exist bugs
- Think more about the review algorithm of SOFL. Recent algorithm about satisfying review, especially when dealing with assertion part of quantified-expression and operator system is fairly complicated. We want to improve it in the future.
- Do more testing on Soflipse, not only for functional requirements, but also some other points such as performance.

References

[1] Eclipse. http://www.eclipse.org.
[2] Abraham Silberschatz, Henry F. Korth, “Database System Concepts”, 4th Ed., McGraw-Hill Education, pp. 210-212, 2006.
[3] Appel, “Modern Compiler Implementation in Java”, Cambridge, University Press, pp. 214-228., 2003.
[4] Diaz G, Pardo JJ and C. F., “Automatic translation of wscdl choreographies to timed automata”, in Lecture Notes in Computer Science, Vol.3670, pp.230–242, 2005.

[5] B. T. Fu X and S. J., "Analysis of interacting bpel web services", in Proc. of 13th International Conference on the World Wide Web, pp.621–630, 2004.
[6] Genxing Yang and Lizhi Cai, "Software Quality Assurance Testing and Evaluating", Tsing Hua University, pp.51-57, 2007.
[7] Hamilton, "Logic for Mathematicians", Cambridge University Press, pp.45-50, 1978.
[8] Verbeek, "Analyzing bpel processes using petri nets", in Proc. of 2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, pp.59–78, 2005.
[9] Hopcroft JE and U. JD., "Introduction to Automata Theory, Languages, and Computation", 3rd Ed., Addison-Wesley: Reading, pp.350-367, 2006.
[10] X. Huang, "The reliability, security and quality in software", Publishing House of Electronics Industry, Beijing, pp. 12-14, 2002.
[11] J.J.Marciniak, "Encyclopedia of Software Engineering", 2nd Ed., Wiley publications, pp. 372-379,1994.
[12] S. Liu, "A property-based approach to reviewing formal specifications for consistency", in Proc. of 16th International Conference on Software Systems Engineering and Their Applications, pp.1–6, 2003.
[13] S. Liu, "An automated rigorous review method for verifying and validating formal specifications", in Proc. of 2nd International Symposium on Automated Technology for Verification and Analysis, pp.15–19, 2004.
[14] S. Liu, "Formal Engineering for Industrial Software Development", Springer, pp. 342-351, 2008.
[15] S. Morimoto, "A Survey of formal verification for business process modeling", in Lecture Notes in Computer Science, Vol.5102, pp.514–522, 2008.
[16] Pressman R.S., "Software Engineering, a practitioner's approach", McGraw-Hill Science/Engineering/Math, pp. 420-432, 2009.
[17] B. L. Salan G and S. M., "Describing and reasoning on web services using process algebra", in Proc. of International Conference on Web Services, pp.43–50, 2004.
[18] W.M.Salimifard, "Petri net-based modeling of workflow systems: an overview", European Journal of Operational Research, Vol.134, pp.664–676, 2001.
[19] A.A.Stolyar, "Introduction to Elementary Mathematical Logic", Dover Publications Inc., pp.53-68, 1970.
[20] W. K. F. B. T and Y. R. A., "Workflow model for chinese business processes, International Journal of Computer Processing of Oriental Languages, pp. 233–258, 2001.
[21] W. van der Aalst, "Verification of workflow nets", in Proc. of 18th International Conference on Application and Theory of Petri Nets, in Lecture Notes in Computer Science, Toulouse, France, Vol.1248, pp.407–426, 1997.
[22] Van der Aalst, M. Dumas and H.M.W., "An approach based on bpel and petri nets (extended version)", in Technical Report BPM-05-25, BPMcenter.org, pp.210-221, 2005.
[23] J. Y. Yang, Q.Tan and F.Liu., "Transformation bpel to cp-nets for verifying web services composition", in Proc. of International Conference on Next Generation Web Services practices, pp.327-330, 2005.
[24] E. Yourdon, "Modern Structured Analysis", 1st Ed., Prentice Hall International Inc., pp.102-122, 1989.