Automatic Transformation from SOFL Formal Specifications to Programs for Software Verification and Testing

Luo, Xiongwen

(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/10.15002/00012876

Automatic Transformation from SOFL Formal Specifications to Programs for Software Verification and Testing

Xiongwen Luo
Graduate School of Computer and Information Sciences
Hosei University
Tokyo, Japan
xiongwen.luo.2b@stu.hosei.ac.jp

***Abstract*— The Structured Object-oriented Formal Language (SOFL) method is developed to overcome the disadvantages of existing formal methods and provide effective techniques for writing formal specifications and carrying out verification and testing. Although it has been applied to system modeling and design in practical and research projects, SOFL has not been widely applied to the industrial software development systems because of the lack of efficient tool support. Aiming at improving the existing SOFL supporting tool and solving the problem that the formal specifications cannot be directly executed, this paper firstly analyzes the relationship between the structures of SOFL formal specifications and C# programs, and then designs and implements the transforming classes for module transformations and data type transformations. Finally, a test is performed to ensure the reliability and validity of the implemented software system.**

Keywords— SOFL; Formal specifications; Automatic transformations; Programs;

I. INTRODUCTION

Formal Methods (FM), consisting of formal specification and formal verification, are of great significance in software systems development. Many formal methods have been reported in the literatures so far, such as VDM [1], Z [2] and B-Method [2]. Although we should not deny that formal methods have many advantages and play a positive role in software engineering, there are several challenges to be resolved in formal methods and their industrial application.

In order to provide an effective way to apply formal methods to industrial software systems, Formal Engineering Method (FEM) was put forward for the first time in 1997 and continued to be used in many publications since then [2]. Furthermore, FEM embraces integrated specification, integrated verification and all kinds of supporting techniques for specification construction, transformation, and system verification and validation [3]. Adopting FEM can reduce the complexity and improve the intuition of formal methods, which provides an effective approach to applying formal methods to industrial software systems, especially for the large-scale and complex software systems.

SOFL is one of the most representative formal engineering methods. Resulting from the integration of Data Flow Diagrams (DFD), Vienna Development Method- Specification Language (VDM-SL) and Petri nets, SOFL has a complete architecture and framework. It integrates structured methods and object-oriented methods, which can offer a way to support functions decomposition and object composition effectively [4].

Owing to the unique and distinctive characteristics, SOFL has many advantages. By combining operations and formal graphic symbols, SOFL creates an exclusive approach to constructing formal specifications accurately and intuitively. It adopts a three-step approach to developing formal specifications. This evolutionary method, starting from an informal specification, through a semi-formal one, finally to a formal specification, can not only moderate the complexity of creating formal specifications, but also improve the intuition and comprehensibility of formal specifications to ensure that the specifications are desirable. On account of its three step approach to developing formal specifications and Condition Data Flow Diagram (CDFD), using SOFL can strike a good balance among visualization, precision and simplicity.

Although SOFL has been applied to system modeling and design in both industrial and research projects [5][6], it has not been widely adopted in the industrial applications owing to the lack of efficient tool support. Aiming at contributing to the development of existing SOFL supporting tool and solving the problem that the formal specifications cannot be directly executed, this paper discusses the transformation from SOFL formal specifications to C# programs, and designs an effective algorithm for building a framework to implement these transformations. After completing the transformation, it serves for specification verification, specification animation and automatic programs testing. Moreover, it also can lay the foundation for automatic test case generation and further researches.

The rest of this paper is organized as follows: Section II analyzes the structure of module and data type in SOFL formal specification and considers how to transform them to programs. Then the design and implementation of transformations are discussed in Section III. Section IV presents the result of

testing the transformation tool. The related work is given in section V. Finally, Section VI concludes the work of this paper and points out the future research directions.

II. TRANSFORMATION PRINCIPLES

In SOFL formal specifications, the module is the most important element, which is regarded as a functional abstraction. And the specifications consist of a collection of corresponding modules in a hierarchical manner. Further, the data type defined in SOFL formal specifications are unique and not identical with the C# program. For the purpose of supporting transformations and utilizing the transformed results to verify specifications and generate programming testing cases, data type transformation is an indispensable part. So, after studying and researching the knowledge of SOFL in-depth, we consider that module transformations and data type transformations are the two important tasks that need to be completed.

A. Module Transformation

In the formal specifications written in SOFL, there are two major parts. One is the CDFD, which indicates the functional behaviors of the integrated processes represented by the graphical symbols. The other is the module, which is an encapsulation of data and process appeared in the CDFD. In general, a module has the structure as follows:

```
module ModuleName / ParentModuleName;
const ConstantDeclaration;
type TypeDeclaration;
var VariableDeclaration;
inv TypeandStateInvariants;
behav CDFD_no;
InitializationProcess;
process ProcessName(input) output;
ext ExternalVariable;
pre PreCondition;
post PostCondition;
decom LowerLevelModuleName;
comment
end_process
    …
Function_1;
Function_2;
    …
Function_m;
end_module
```

Fig. 1. Structure of module

The beginning of the module is the keyword **module**. *ModuleName* is an unique identifier of module in SOFL specifications. Since the module describes a decomposition of a high level process, the name of module should include the *ParentModuleName* which is the higher level module. Then the key word **const**, **type** and **var** start the parts for constant declarations, type declarations, and variable declarations respectively. The key word **inv** stands for the type and state invariants, which represents the constraints on the type declarations section and variable declarations section. The *CDFD_no* after the key word **behav** specifies the affiliated CDFD. The last two parts, beginning with keyword **process** and **function**, offers some operations and functions.

The process, basically, consists of five parts: process name, input data flow variables, output data flow variables, precondition and postcondition. The process presents an action or operation that consumes the input data flows and generates the output data flows. If there are external variables that need to be used in this process, they are stated after the keyword **ext**. A complex process may be decomposed into the lower level CDFD whose associated module is written after the keyword **decom**. The keyword **comment** starts the informal comment section, which is usually written to improve the readability of the formal specifications.

Through analyzing the specific structure of each part in the module in detail, we consider that the structure of a module is similar to that of a class in C# program. Thus, it is quite natural and convenient to transform a module to a C# class. Several underlying guidelines proposed for module transformations are presented in Table I:

Table I Main idea of module transformation guidelines

SOFL Module	C# language
module M	class M
const T X	const T X
Type	state a type inner class
var y	variables in external files
process initialization	constructor method
process A	method A
function F	method F

In summary, the details about module transformation guidelines shown in Table I are following:

- Transform the module name to the corresponding class name.
- Transform the constant declaration to the constant in C#, using the keyword **const** prior to the constant variables.
- Transform the type declaration to either a basic type or a class, whose form is in compliance with C# language syntax.
- Transform the variable declarations to the instance variables, stored and accessed in the external file.
- Transform the processes to the target methods.
- Transform the functions to the target methods, which are similar to that of processes.

B. Data Type Transformation

Data types, an essential part of SOFL formal specifications, provide notations to define data structures in the SOFL formal specifications. Because the data types of SOFL are not identical with C# data types both in semantics and syntax, we cannot directly execute the results of module transformations. In other words, it is attached no significance to the

transformations of module without the data type transformations. Only with the supporting of data types transformations, results of module transformations can be used for specifications and programming testing.

In SOFL, the data types are classified into two categories: built-in types and user-defined types. The built-in types have fourteen kinds of data types, which are further divided into basic types and compound types. Transformations from built-in data types in SOFL to the data types in C# require both semantics preservation and syntactic changes [7]. The syntax of variable declaration in SOFL is not identical with that in C#. The former lets the type appear after the variable with a colon separating them, while the latter makes the type appear prior to the variable with a space between them. In the semantics perspective, some of the data types in SOFL and C# are similar, while some of the others are totally different. The user-defined types are defined by the specification writers. They are based on the built-in data types, so the transformation guidelines of built-in data types also apply to the transformation of user-defined types.

In general, the choice of the concrete data types in the transformation will affect somehow the algorithms of the implemented program using the data types [8]. Therefore, it is essential to strike a balance between data structures and algorithms. An executable outline about transformations of all the built-in data types in SOFL formal specifications are presented in Table II:

Table II Transformation of data types

SOFL data type	C# data type
int	int
char	char
string	string
bool	bool
Enumeration	enum
nat0	int
nat	int
real	double
set	HashSet
seq	List
map	Dictionary
composite	abstract class
product	abstract class
union	class

Furthermore, there are several principles for data type transformation summarized as follows:

- The *int* type, *char* type, *string* type, *bool* type, and *Enumeration* type do not need to be transformed because they have already existed in C# and can be used directly.
- The *nat0* type defines the natural numbers including zero and *nat* type defines the natural numbers, so that these two types can be implemented by *int* type in C#.
- The *real* type represents the real numbers, which can be implemented by *double* type in C#.
- The *set* type is an unordered collection of distinct objects, which is similar to the *HashSet* type in C#, so it is natural to implement it through *HashSet* type.
- The *sequence* type is an ordered collection of objects that allows duplications of objects. Taking this into account, I believe that *List* type in C# is the best choice to implement it.
- The *map* type is a finite set of pairs, the domain and range to the map share the similar meaning with the key and value to the D*ictionary* type in C#.
- The *composite* type and *product* type represent a collection of several data items, so the abstract classes are used to implement these two types and their inherent functions.
- The *union* type is a special type associated with several functions. It can be regarded as a collection of variables in different types. We consider that we will transform this type to a class with many fields in C#.

III. Design and Implementation

A. Abstract Tree of Transformation

In general, the transformations can be divided into module transformations and data type transformations. The module transformations would be broken down to lower level transformations, while the data type transformations are also composed of several lower levels transformations. In this case, a tree structure has been built to clearly and correctly describe the top-down decomposition of transformations. For this kind of tree structure, we propose an abstract tree of transformations [9-11] shown in Fig. 2.

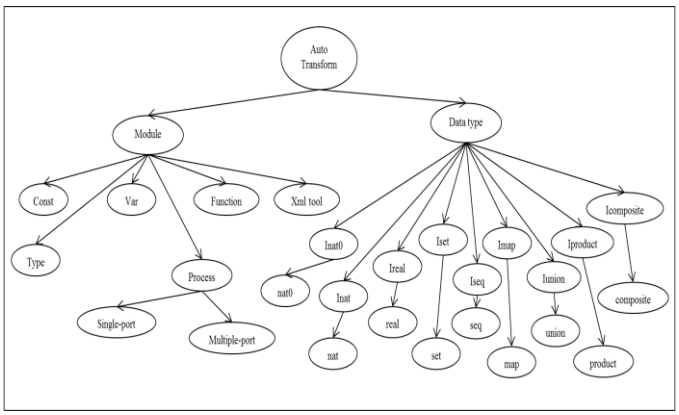


Fig. 2. Abstract tree of transformations

B. Transformation Rules and Algorithms

Some principles of the abstract tree related to the transformations are as follows:

- As for the overall transformations, the main program should be corresponding to the root node of the abstract tree.

- The module transformations and data type transformations are the two child nodes of the root node.
- The module transformations node include the constant transformation node, type transformation node, variable transformation node, process transformation node, function transformation node and XML tool node.
- Except for the process transformation node, other nodes are similar. The process node can be divided into single-port process child node and multiple-port process child node. Other nodes are the corresponding terminal nodes.
- The data type transformations node contains nine data type interface child nodes that need to be transformed, each of which has a terminal node representing the implementation class corresponding to the related data type interface.

Based on the above principles, starting from the top transformation, a root node is created, namely the automatic transformation node. And then it extends to the child nodes step by step, until the terminal node is built.

Through using preorder traversal algorithm, we can complete every part of the transformations. The abstract tree shown in Fig. 2 offers an outline of the transformations and can help us clearly understand the complete structure of the transformations. It is also a necessary part to check the integrity of the programs and ensure the reliability and correctness of the whole systems.

C. Design and implementation of classes

In general, we create three packages to implement the transformations. One is automatic transformation package, and other two are module transformations package and data type transformations package as shown in Fig. 3. In the automatic transformation package, we need to invoke the methods defined in the module transformations package to complete the module transformations. The results of transformations cannot be executed without the support of data type transformations which are completed in SOFL date type package.

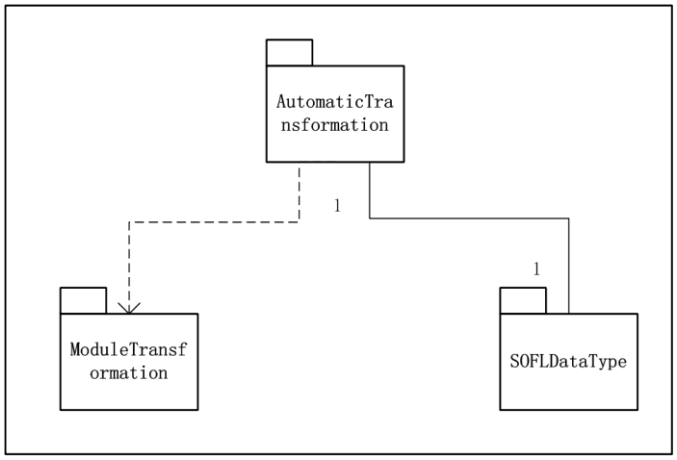


Fig. 3. Structure of transformation framework

1) Implementation classes in module transformation package

The module transformation package involves several classes as follows:

- XmlTool class: the objective of this class is to provide a XML file tool used for extracting the data information of SOFL formal specifications form the XML files. Before executing the transformations process, we should make formal specifications generate the corresponding XML files through the existing SOFL supporting tool, then this class is to parse these XML files to get the data information we need.
- ModuleDeclarationTransformation class: this class is to realize the functions of module transformations by invoking the methods in other classes. It has two methods, one is to write the first line of module and the other is to complete the transformations of other parts in module sequentially.
- ConstantTransformation class: this class mainly deals with constant declaration in SOFL formal specifications. It contains two methods, one is to get the constant variables and write into the external file, and the other is to judge constant variables types and invoke the former method to write the corresponding constants.
- TypeTransformation class: this class is used to complete the type declaration transformations. Because there are many kinds of different data types, we need to invoke different methods to implement the transformations. In other words, each of the compound type transformation uses one method.
- VariableTransformation class: The purpose for this class is to transform the variable declaration section to the programing in C#. Since the variables appearing in this corresponding part are either the local variables or external variables, so we design two methods to implement this process. One is for writing the local variables into target files and the other is for writing the external variables.
- ProcessTransformation class: In this class, a method is defined to judge different cases, and for each case, we invoke the methods of SinglePort class to execute the single-port process transformations and the methods of MultiplePort class to exe-cute the multiple-port process transformations.
- FunctionTransformation class: this class is to handle the transformations from function declaration to programs. Owing to the similar structure with the method in C#, we design a method to write these function declarations into the target files.

2) Implementation classes in SOFL data type package

There are fourteen kinds of data types in SOFL, but five of them share the same semantics and syntactic with C# language so that they can be directly executed in the programming. The rests need to be transformed in the C# to support the results of module transformations. In this case, we design nine interfaces

to implement the transformations of nine kinds of data types. The relationship is shown in Table III:

Table III SOFL data types and implementation classes

SOFL data type	Data type interface	Implementation class
nat0	Inat0	nat0
nat	Inat	nat
real	Ireal	real
set	Iset	set
seq	Iseq	seq
map	Imap	map
composite	Icomposite	composite
product	Iproduct	product
union	Iunion	union

Note that the naming conventions of class in C# is that the first letter of class name is capitalized, but we do not observe this rule because we want to make the name of implementation classes in accordance with the keyword of data type in SOFL, so as to use the implementation classes efficiently and unambiguously.

In the SOFL data type package, We design nine interfaces, which are *Inat0*, *Inat*, *Ireal*, *Iset*, *Iseq*, *Imap*, *Icomposite*, *Iproduct* and *Iunion*, to implement the transformations of these nine kinds of data types. The methods in each interface are in accordance with the operators in the related SOFL data types. Then nine classes, which are *nat0*, *nat*, *real*, *set<T>*, *seq<T>*, *map<T, E>*, *composite*, *product* and *union*, are created to implement the corresponding interfaces.

3) Main program in automatic transformation package

The automatic transformation package includes the main class of the transformations, whose *main method* is the entry of the automatic transformations. This package invokes the methods of classes in the module transformation package and is supported by the classes in the SOFL data type package.

In this process, firstly, we enter the path of XML file, and then judge whether the file exists or not. If the XML file exists, we have to enter the output file path and also make a judgement to ensure that the file name is legal. In the next step, we will make a choice to decide whether starting the transformations or not. If we choose to start the transformations, the specifications will be transformed to C# programs. After completing the transformation, the system is exited.

IV. TRANSFORMATION RESULTS

After the transformation software system is implemented, it is essential to perform a test to detect faults and ensure the validity and robustness of the system. In order to check whether each function in the transformations can be used correctly, we adopt the black-box testing method to test the transformations process. Firstly, unit testing method is used to test each transformation section, namely the constant transformation, type transformation, variable transformation, process transformation and function transformation. And then the integration testing and system testing methods are adopted to ensure the success of the entire formal specification transformations.

The testing procedures can also be the guidance of how to use these programs to make the automatic transformation, which are listed as follows:

Step 1: use the existing SOFL supporting tool to create the formal specifications and draw the related CDFDs [12].

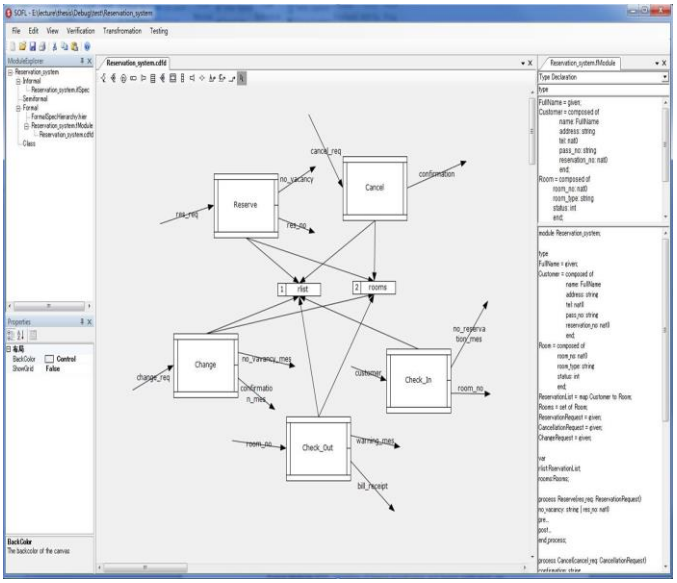
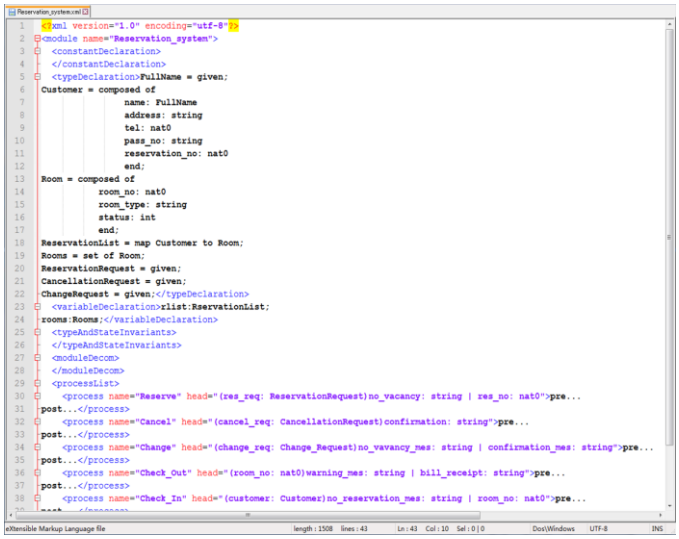


Fig. 4. SOFL formal specification of ModuleTest

In the existing SOFL supporting tool, we can use the three-step approach to constructing the formal specifications as shown in Fig. 4. The structure of the components in current project is displayed in the upper-left corner. In the center, a CDFD related to the module is drawn. If one item in the CDFD is selected, the attributes of it will be presented in the lower-left corner. The module in detail is written in the right side.

Step 2: generate the related XML file through the existing SOFL supporting tool.



```
<?xml version="1.0" encoding="utf-8"?>
<module name="Reservation_system">
  <constantDeclaration>
  </constantDeclaration>
  <typeDeclaration>FullName = given;
Customer = composed of
             name: FullName
             address: string
             tel: nat0
             pass_no: string
             reservation_no: nat0
             end;
Room = composed of
          room_no: nat0
          room_type: string
          status: int
          end;
ReservationList = map Customer to Room;
Rooms = set of Room;
ReservationRequest = given;
CancellationRequest = given;
ChangeRequest = given;</typeDeclaration>
  <variableDeclaration>rlist:ReservationList;
rooms:Rooms;</variableDeclaration>
  <typeAndStateInvariants>
  </typeAndStateInvariants>
  <moduleDecom>
  </moduleDecom>
  <processList>
    <process name="Reserve" head="(res_req: ReservationRequest)no_vacancy: string | res_no: nat0">pre...
post...</process>
    <process name="Cancel" head="(cancel_req: CancellationRequest)confirmation: string">pre...
post...</process>
    <process name="Change" head="(change_req: Change_Request)no_vavancy_mes: string | confirmation_mes: string">pre...
post...</process>
    <process name="Check_Out" head="(room_no: nat0)warning_mes: string | bill_receipt: string">pre...
post...</process>
    <process name="Check_In" head="(customer: Customer)no_reservation_mes: string | room_no: nat0">pre...
```

Fig. 5. XML file of related SOFL formal specification

In Fig. 5, the names of labels are related to the corresponding keywords in the SOFL formal specification constructed in step 1. For example, the label "module" is related to the keyword **module** in the specification.

Step 3: Using the software system we have developed to parse the XML file and complete the transformation. A result of the transformation is presented in Fig. 6.

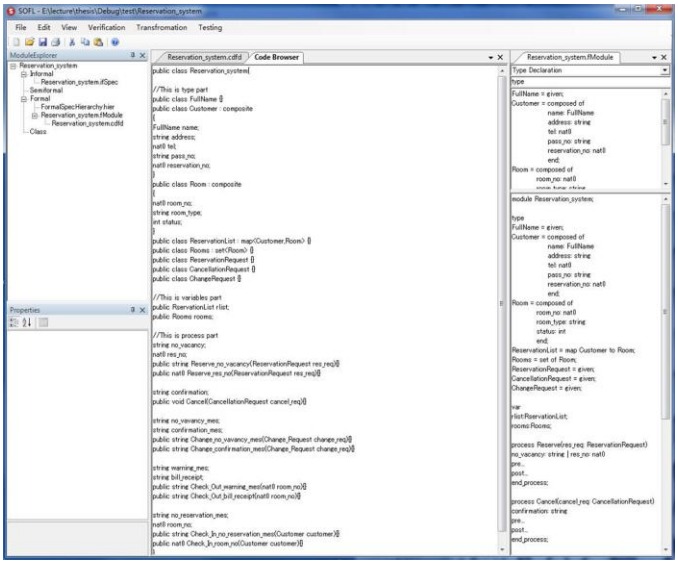


Fig. 6. Results of transformations

Using the implemented software system, we can parse the XML files corresponding to the SOFL formal specifications and transform them into C# programs. In Fig. 6, the module name is related to the class name. The constant declarations in the module are transformed to the constant variables. The type declarations are transformed to either a basic type or a class. The variable declarations are transformed to the instance variables. The process and function are implemented by the target methods.

V. Related Work

There exist some tools to support automatic transformation from other formal notation to programming languages. VDMTools reported in [13] offers the functions of analyzing system models expressed in the formal notation VDM-SL, which has been successfully applied to developing industrial software systems. The VDM specification can be executed directly through the interpreter inside this tool. User can test the VDM specification by providing test cases and observe the system behavior by setting breakpoints or stepping. ProB[14] is a validation toolset for the B method. In this tool, a model checker and a refinement checker can be used for executing the B specifications to detect various errors. However, in order to perform the exhaustive model checking, the given sets must be finite, and the integer variables must be restricted to a small range. UPPAAL[15] is a verification tool for timed automate, which allows user to model the system behavior in terms of states and transitions between states. In UPPAAL, there is no specification written in words, but user can construct the finite state machine to module the functions of system, which can be executed in this tool to detect faults.

VI. Conclusions and Future Work

In this paper, aiming at contributing to the development of existing SOFL supporting tool and making the automatic transformation from SOFL formal specifications to programs for software verification and testing, we discuss the implementation of transformations from SOFL formal specifications to C# programs. Firstly, we analyze the module structure and data type in the SOFL formal specifications, which lays the foundation for the transformation. Then, the design and implementation of the transformations are described. After implementing the transformations systems, we use black-box testing method to verify the results of transformations. We believe that it is a vital part to make the transformations and it can serve for the specification verification and automatic generation of testing cases, which are very useful and meaningful in software development.

In the future, we will continue the transformation work and plan to extend to the CDFD and class in SOFL formal specifications. With the development of transformations, we are also interested in applying the transformation results to specifications testing and specification animation.

References

[1] Peter D.Mosses, "VDM semantics of programming language: combinators and monads," Formal Aspects of Computing, 2011, 23: 221-238.

[2] Shaoying Liu, "Formal engineering for industrial software development using the SOFL method", Berlin Heidelberg: Springer-Verlag, 2004.

[3] Shaoying Liu, "An approach to applying SOFL for agile process and its application in developing a test support tool," Innovations Syst Softw Eng, 2010, 6:137-143.

[4] Fauziah binti Zainuddin, Shaoying Liu, "An approach to low-fidelity prototyping based on SOFL informal specification", IEEE APSEC, 2012, 1530-1362/12.

[5] Weikai Miao, Shaoying Liu, "Service-Oriented modeling using the SOFL formal engineering method", IEEE APSCC, 2009, 978-1-4244-5336-8/09.

[6] Shaoying Liu, Xiang Xue, "Automated software specification and design using the SOFL formal engineering method", IEEE WCSE, 2009, 978-0-7685-3570-8/09.

[7] A Rahman Mat, Cheah Wai Shiang, Shaoying Liu, "SOFL three-step approach to construct the formal specification of a brain tumor treatment system", SDIWC, 2013, ISBN: 978-0-9853483-3-5.

[8] Yuting Chen, "A case study of using SOFL to specify a concurrent software system", IEEE, 2010, 978-1-4244-6055-7/10.

[9] Jie Han, Haopeng Chen, "Requirements analysis based on SOFL formal method, Computer Applications and Software", 2007, 24(9):57-59.

[10] Xiaoli Fang, Haopeng Chen, "Principle and realization of review based of SOFL specification", Computer Engineering, 2006.9, 32(8).

[11] Zhenghua Gao, Haopeng Chen, "Semantic analysis based on SOFL specification", Computer Applications and Software, 2007.11, 24(11).

[12] Mo Li, Shaoying Liu, "Tool support for rigorous formal specification inspection", IEEE CSE, 2014, 978-1-4799-7981-3/14.

[13] John Fitzgerald, Peter Gorm Larsen, Shin Sahare, "VDMTools: Advances in Support for Formal Modeling in VDM", ACM SIGPLAN Notices, 43(2), 2008.

[14] Michael Leuschel, Michael Butler, "PROB: An Automated Analysis Toolset for the B Method", Int J Softw Tools Technol Transfer, 10:185-203, 2008.

[15] Gerd Behrmann, Alexandre David, Kim G.Larsen, "UPPAAL 4.0", IEEE QEST06, 0-7695-2665-9/06, 2006.