Shaoying Liu (Ed.)

LNCS 7787

Structured Object-Oriented Formal Language and Method

Second International Workshop, SOFL 2012
Kyoto, Japan, November 2012
Revised Selected Papers

Lecture Notes in Computer Science 7787

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Shaoying Liu (Ed.)

Structured Object-Oriented Formal Language and Method

Second International Workshop, SOFL 2012
Kyoto, Japan, November 13, 2012
Revised Selected Papers

Springer

Volume Editor

Shaoying Liu
Hosei University
Faculty of Computer and Information Sciences
Department of Computer Science
3-7-2 Kajino-cho Koganei-shi
Tokyo 184-8584, Japan
E-mail: sliu@hosei.ac.jp

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39276-4 e-ISBN 978-3-642-39277-1
DOI 10.1007/978-3-642-39277-1
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013946042

CR Subject Classification (1998): F.3, D.2, D.2.4, D.3, F.4.1, F.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

There is a growing interest in applying formal methods in practice to improve software productivity and quality, but only with a few exceptions, this interest has not been successfully converted into reality. How to enable practitioners to easily and effectively use formal techniques still remains challenging.

The Structured Object-Oriented Formal Language (SOFL) has been developed to address this challenge by providing a comprehensible specification language, a practical modeling method, various verification and validation techniques, and tool support through effective integration of formal methods with conventional software engineering techniques. SOFL integrates data flow diagram, Petri nets, and VDM-SL to offer a visualized and formal notation for writing specifications; a three-step approach to requirements acquisition and system design; specification-based inspection and testing methods for detecting errors in both specifications and programs; and a set of tools to support modeling and verification.

The WSOFL 2012 workshop was organized by the Shaoying Liu research group at Hosei University with the aim of bringing industrial, academic, and government experts of SOFL together to communicate and to exchange ideas. The workshop attracted 19 submissions on formal specification, specification-based testing, specification pattern, integration of formal specification and prototyping, specification animation, application of SOFL, and supporting tools for SOFL from over five countries. Each submission was rigorously reviewed by at least two PC members or their co-reviewers on the basis of technical quality, relevance, significance, and clarity, and 10 papers were accepted for publication in the workshop proceedings. The acceptance rate is about 52.6%. The workshop program featured a mini tutorial on SOFL and 11 paper presentations. Around 30 people from both industry and academy participated in the workshop.

I would like to thank the ICFEM 2012 organizers for accepting our proposal and supporting the organization of the workshop, all of the program committee members for their great efforts and cooperation in reviewing and selecting papers, and my PhD students Xi Wang and Weikai Miao for their help in setting up the EasyChair account and the homepage for the workshop. I would also like to thank all of the attendees for their active participation in discussions at the workshop. Finally, my gratitude goes to Alfred Hofmann and Anna Kramer of Springer for their support in the publication of the workshop proceedings.

Shaoying Liu

Organization

Program Committee

Shaoying Liu (Chair)	Hosei University, Japan
Michael Butler	University of Southampton, UK
Steve Cha	Korea University, Korea
Jian Chen	Shaanxi Normal University, China
Yuting Chen	Shanghai Jiaotong University, China
Jin Song Dong	National University of Singapore
Mo Li	Hosei University, Japan
Abdul Rahman Mat	University Malaysia Serawak, Malaysia
Weikai Miao	Hosei University, Japan
Fumiko Nagoya	Aoyama Gakuyin University, Japan
Jeff Offutt	George Mason University, USA
Shengchao Qin	University of Teesside, UK
Jing Sun	University of Auckland, New Zealand
Cong Tian	Xidian University, China
Xi Wang	Hosei University, Japan
Fauziah Zainuddin	Hosei University, Japan
Hong Zhu	Oxford Brookes University, UK
Wuwei Shen	Western Michigan University, USA

Additional Reviewers

Chung-Ling Lin
Yanhong Huang

Table of Contents

Testing and Tools

Applying "Functional Scenario-Based" Test Case Generation Method in Unit Testing and Integration Testing 1
Cencen Li, Mo Li, Shaoying Liu, and Shin Nakajima

Supporting Tool for Automatic Specification-Based Test Case Generation 12
Weihang Zhang and Shaoying Liu

A Formal Specification-Based Integration Testing Approach 26
Weikai Miao and Shaoying Liu

Tools for Specification

Design and Implementation of a Tool for Specifying Specification in SOFL 44
Mo Li and Shaoying Liu

Development of a Supporting Tool for Formalizing Software Requirements 56
Xi Wang and Shaoying Liu

Model Checking

Abstract Model Checking with SOFL Hierarchy 71
Cong Tian, Shaoying Liu, and Zhenhua Duan

Model Checking C Programs with MSVL 87
Yan Yu, Zhenhua Duan, Cong Tian, and Mengfei Yang

Application and Prototyping

An Application of SOFL for Rapid Prototyping 104
Fumiko Nagoya and Tetsuo Kitagawa

Applying SOFL to a Generic Insulin Pump Software Design 116
Chung-Ling Ling, Wuwei Shen, and Dionysios Kountanis

Extension on Transactional Remote Services in SOFL 133
Yisheng Wang and Haopeng Chen

Author Index 149

Applying "Functional Scenario-Based" Test Case Generation Method in Unit Testing and Integration Testing*

Cencen Li[1], Mo Li[1], Shaoying Liu[2], and Shin Nakajima[3]

[1] Graduate School of Computer and Information Sciences, Hosei University, Tokyo, Japan
{cencen.li.js,mo.li.3e}@stu.hosei.ac.jp
[2] Department of Computer and Information Sciences, Hosei University, Tokyo, Japan
sliu@hosei.ac.jp
[3] Information Systems Architecture Science Research Division, National Institute of Informatics, Tokyo, Japan
nkjm@nii.ac.jp

Abstract. Specification-based testing enables us to detect errors in the implementation of functions defined in given specifications. Its effectiveness in achieving high path coverage and efficiency in generating test cases are always major concerns of testers. The automatic test case generation approach based on formal specifications proposed by Liu and Nakajima is aimed at ensuring high effectiveness and efficiency, but this approach has not been used under different testing environments. In this paper, we first present the statical analysis of the characteristics of the test case generation approach, and then show the experiments of using this approach in two different real testing environments. The two practical testing cases include a unit testing and an integration testing. We perform the testing not only for assessing Liu's approach in practice, but also trying to get some experience of using this approach in practice. The static analysis and the results of experiments indicate that this test case generation approach may not be effective in some circumstances, especially in integration testing. We discuss the results, analyze the specific causes for the ineffectiveness, namely the low path coverage, and propose some suggestions for improvement.

Keywords: experiment, functional scenario, specification-based, testing.

1 Introduction

Specification-based testing enables us to detect errors in the implementation of the functions defined in given specifications. Since performing specification-based

* This work is supported by NII Collaborative Research Program. Shaoying Liu is also supported by the NSFC Grant (No. 60910004), and 973 Program of China Grant (No. 2010CB328102).

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 1–11, 2013.

testing is usually time consuming, automatic specification-based testing is always attractive to the software industry. An automatic test cases generation approach based on formal specification known as *functional scenario-based testing*(FSBT) was first introduced in Liu's paper [1], which is aimed to ensure high effectiveness and efficiency of specification-based testing.

This automatic specification-based testing approach is applicable to any formal specifications that are comprised of operations specified in terms of pre- and post-conditions. The essence strategy underlying this testing approach is to guarantee that every functional scenario defined in a specification is implemented "correctly" by the corresponding program. A functional scenario of an operation defines an independent relation between its input and output under a certain condition, and usually expressed as a predicate expression. The predicate expression is used as a foundation of automatic test case generation algorithm. The function defined by a functional scenario is actually a function of the software system, and it should be implemented in the program. Therefore, by using the test cases derived from functional scenarios, the implementation of functions defined in scenarios are expected to be tested consequently.

To assess this testing approach formally, we performed a static analysis and an experiment in out previous publication [2]. The static analysis is carried out to analyse the characteristics of the testing approach. We focused on uncovering the relationship between functional scenario and corresponding execution path. Based on the definition of functional scenario, it presents a specific function of a system. Correspondingly, there should be an execution path in the program to implement the function defined by the functional scenario. We separated the relationship into five categories, and pointed out the circumstance in which each kind of relationship occurs. The details will be briefly introduced in Section 3.

In this paper, we present two experiments of using the test case generation method in different testing environments. One of the experiment is a unit testing, and the other one is an integration testing. The details of the integration testing have been presented in [2]. Here we discuss its results with the result of the unit testing together. We compare and analyse the results and the procedures of these two experiments. The results of experiments indicate that the testing approach may have a high level path coverage in unit testing due to using lower level specification to generate test cases. But the effectiveness of testing will be reduced if the specification is not well-formed [3], or the function defined by functional scenario is refined by programmers in implementation. This indication confirms the conclusion of statical analysis practically. We totally performing two experiments not only for assessing the test case generation method in practice, but also trying to get some experience of using this method in practice.

The remainder of this paper is organized as follows. Section 2 includes brief introduction of the original decompositional testing approach, including the test strategy and the test case generation algorithm. In Section 3 we describe the relations between functional scenarios and execution paths in program, which are two basic concepts in our experiment. The experiment will be introduced in Section 4 including purpose, environment, results and results analysis. We will

Table 1. Test Cases Generation Algorithm

No. of Algorithms	$\ominus$	Algorithms of test case generation for x_1
1	$=$	$x_1 = E$
2	$>$	$x_1 = E + \Delta x$
3	$<$	$x_1 = E - \Delta x$
4	$\leq, \geq, \neq$	similar to above

propose suggestions based on our experience in Section 5. Section 6 is related work, and in Section 7 we conclude the paper and point out future research directions.

2 Decompositional Approach to Automatic Test Case Generation

In order to explain the test case generation algorithm, we need to define the formal specification and functional scenario first. For simplicity, let $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ denote the formal specification of an operation S, where S_{iv} is the set of all input variables, S_{ov} is the set of all output variables, and S_{pre} and S_{post} are the pre and post-condition of S, respectively. For the post-condition S_{post}, let $S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee ... \vee (C_n \wedge D_n)$, where each $C_i (i \in \{1, ..., n\})$ is a predicate called a "*guard condition*" that contains no output variable in S_{ov} and $\forall_{i,j \in \{1,...,n\}} \cdot i \neq j \Rightarrow C_i \wedge C_j = false$; D_i a "*defining condition*" that contains at least one output variable in S_{ov} but no guard condition. Then, a formal specification of an option can be expressed as a disjunction expression $(\sim S_{pre} \wedge C_1 \wedge D_1) \vee (\sim S_{pre} \wedge C_2 \wedge D_2) \vee ... \vee (\sim S_{pre} \wedge C_n \wedge D_n)$. A conjunction $\sim S_{pre} \wedge C_i \wedge D_i$ is realized as a functional scenario. We treat a conjunction $\sim S_{pre} \wedge C_i \wedge D_i$ as a functional scenario because it defines an independent behavior: when $\sim S_{pre} \wedge C_i$ is satisfied by the initial state (or input variables), the final state (or the output variables) is defined by the defining condition D_i.

Since test case generation usually depends on the pre-condition and guard condition, and the defining condition D_i usually does not provide the main information for test case generation. The defining condition D_i is eliminated from the functional scenario. The conjunction after eliminating defining condition is $\sim S_{pre} \wedge C_i$, called *testing condition*. For each atomic predicate Q in testing condition, the input variables involved in each atomic predicate expression Q can be generated by using an algorithm that deals with the following three situations, respectively.

- **Situation 1:** If only one input variable is involved and Q has the format $x_1 \ominus E$, where $\ominus \in \{=, <, >, \leq, \geq, \neq\}$ is a relational operator and E is a constant expression, using the algorithms listed in Table 1 to generate test cases for variable x_1.

- **Situation 2:** If only one input variable is involved and Q has the format $E_1 \ominus E_2$, where E_1 and E_2 are both arithmetic expressions which may involve variable x_1, it is first transformed to the format $x_1 \ominus E$. And then apply Criterion 1.
- **Situation 3:** If more than one input variables are involved and Q has the format $E_1 \ominus E_2$, where E_1 and E_2 are both arithmetic expressions possibly involving all the variables $x_1, x_2, ..., x_w$. First randomly assigning values from appropriate types to the input variables $x_2, x_3, ..., x_w$ to transform the format into the format $E_1 \ominus E_2$, and then apply Situation 2.

Note that if one input variable x appears in more than one atomic predicate expressions, it needs to satisfy all the expressions which it is involved in.

3 Static Analysis

The objective of program testing is to test all parts of the program. This objective indicates that all of the execution paths in the program should be tested. An execution path can be realized a sequence of statements from the start state of the program to the termination. Based on the definition of functional scenario, an execution path can be realized as an implementation of a functional scenario. Theoretically, one functional scenario should correspond to one and only one execution path if the program is implemented by following the formal specification exactly. But in practice, the relation between functional scenario and execution path may not be a one-to-one correspondence. In order to figure out how the test cases derived from a functional scenario influence the coverage of execution paths, we define that a scenario and a execution path have relation to each other if all of the test cases derived from the scenario can be accepted by the execution path. Although this definition is not sufficient to describe various relations between scenarios and paths, it is enough for the purpose of our experiment. The summary of the relations between functional scenarios and execution paths are listed in the following.

- **One Scenario to No Path:** If the function defined by the functional scenario is not implemented in the program or implemented incorrectly, there is no path being executed by applying the test cases derived from the scenario. It may be caused by the programmer misunderstanding the specification, or mistake made by the programmer during programming.
- **One Scenario to One Path:** This is the ideal situation, the program is implemented according to the specification exactly. No refinement is made in the specification or program.
- **One Scenario to Multi Paths:** This is the most common situation. It is usually caused by the refinement, which may occur in specification or program. Since some specifiers use top-down approach when they specify specifications, they will define the more general or more abstract process with less details first, and then decompose the process into more than one lower

level process with more details. When we try to find execution paths for the functional scenarios extracted from a more abstract level specification, it is possible to find more than one paths corresponding to one specific scenario if the program is implemented based on the lower level specification. If the information in lower level specification is not considered in test cases generating process, some of the relative paths will not be tested. Another kind of refinement occurs in the program. It is usually made by the programmer for different reasons, like improving the effectiveness of program, complying with the special programming rules, etc.

- **Multi Scenario to One Path:** The relation multi scenarios to one path is a reverse relation of one scenario to multi paths, it usually occurs when programmer abstracts some functions defined in the specification. The best reason for programmer to abstract the function is to simplify the program.
- **Paths to No Scenario:** This situation is very common in practice, but unfortunately it will often be ignored in specification-based testing. According to the concept of specification-based testing, the process of testing is on the basis of what the specification says. But, in the real testing environment, even all of the execution paths which implement all of the defined functions are tested, it does not mean that all parts of the program have been tested. The most possible reason of the occurrence of this kind of relation is the incompleteness of specification. The incompleteness can be caused by lacking ideas or limitation of time, etc. But, in the meantime, the programmer may try to, or have to, handle some exceptions or add some functions undefined in the specification. One specific case is that the program needs to process the input variables even the values of the input do not evaluate the predicates of the scenarios to be true. This is because the specification just defines what kind of inputs can be handled while the program must respond to all of the possible inputs. Usually we think these kind of paths *relative to process.*

4 Experiments

In this section, we present two experiments. Each experiment performs a testing by applying the functional scenario-based test case generation method. One of the experiments is unit testing, and the other is an integration testing. Carrying out these two testing experiments is not aim to compare the effectiveness of the test cases generation method among specific cases, but to assess the effectiveness of the method under different test environments and attain some experience in using the method in practice.

In our cases, if the functional scenarios used to generate test cases are from a lower level module, the testing can be realized as a "unit testing". On the contrary, if the functional scenarios used to generate test cases are from a higher level module, we consider the testing as an "integration testing". Different researchers may have different understanding under their viewpoint. Here we just

use the concepts of unit testing and integration testing to different two kind of testing cases, one is based on the relatively higher level specification and the other one is based on the relatively lower level specification.

Since our major concern in the experiment is the effectiveness of the testing approach or how many parts in the program of target system can be tested, we use the coverage of the executable paths in the program to measure the effectiveness of the testing approach.

4.1 Target Systems

Income Tax Calculation System. The Income Tax Calculation system is the target system of the unit testing. This system is aim to help tax payers to calculate their amount of tax. According to [13], the tax payers are divided into two categories based on the type of their income. The two categories of tax payers use different formulas to calculate the amount of tax, but they have the similar process. The tax payer first calculates his or her "amount of income", and then calculates the"deduction of income". Finally, the "amount of tax" is calculated based on the the difference between "amount of taxable income" and "deduction of income".

The specifications of the Income Tax Calculation system are separated into three levels. The top level, or the first level, specification contains 2 processes. Each process, which presents a tax calculation process for one category of tax payers, is decomposed into a module containing 3 processes in the second level specification. These 3 processes in each second level module correspond to the three steps in tax amount calculation mentioned previously. Each process in the second level modules is decomposed to construct the lowest level, or the third level, specification. There are 61 processes contained in the lowest level specification. And totally 69 processes are defined in all three level specifications, which are formally specified in SOFL. The implementation of the system is developed by using Java under the Eclipse environment. The implementation program contains 15 classes, and more than 2500 lines code.

IC Card System. The target system of the integration testing generating test cases from specification directly is an IC card system. The IC card can be used to take the public transportations, and it associates with a bank account. Since card holders can use the card without authority, the maximum amount that can be deposited in the IC card is limited to prevent the potential economic loss from losing the card. Customers can swipe the cards to take transportation or use the cards to buy train tickets. If the balance in card is not enough, the customers can reload by cash or from associated banks account. The customers can also transfer the money back to the bank accounts, but they can not get cash from the IC cards directly. For the customers who need commute, they can set monthly payment for one route to get discounts.

We design and define 6 processes in the top level module. This module presents the most abstract definition of the IC card system, and each of the 6 processes represents a function of the system described previously. Based on the top-down concept, we decompose each of these processes for further defining. There are total 12 lower level processes defined in the specification and some of them are reused to construct higher level processes. All of these 18 processes are specified formally by using SOFL, and the implementation program contains 14 classes, 2200 lines code.

4.2 Unit Testing

The Income Tax Calculation system is the target system of the unit testing. To perform the testing, we derive test cases from the processes defined in the lowest level specification. Totally 29 processes derived from the same process in the first level specification are used to generated test cases for testing their corresponding program units. Total 615 test cases are generated from the 207 functional scenarios, which are extracted from the 29 processes in the testing. The target program units contain 223 execution paths, and 211 paths are tested by applying the 615 test cases. The average path coverage is approximately 94 percent. The excerpt of the testing results are listed in Table 2. Item "***Number of Scenario***" identifies the scenario in a process; "***Relative Paths***" shows how many execution paths in program have relations with the scenarios; "***Test Cases***" denotes how many test cases are generated from this scenario and "***Tested Paths***" indicates how many paths are tested; "***Coverage***" shows the coverage of executable paths.

Table 2. Excerpt of Unit Testing Result

Process Name	Number of Scenario	Relative Paths	Test Cases	Tested Paths	Coverage (%)
IFDSTAT_A	2	2	8	2	100
EI_A	11	12	35	11	91
MI_A	20	24	63	20	83
OI_A	2	2	8	2	100
T_A	1	3	1	1	100

4.3 Integration Testing

The target system of the integration testing is the IC card system. Only the functional scenarios extracted from the 6 processes in the top level are used to generate test cases. There are 112 execution paths in the program correspond to the functions defined in the 6 processes. Total 192 test cases are generated from the 17 functional scenarios in the testing, and the brief statistics of the results are listed in Table 3. The details of this integration testing can be found in [2].

Table 3. Integration Testing Result

Process Name	Number of Scenario	Relative Paths	Test Cases	Tested Paths	Coverage (%)
RailwayTravel	5	12	35	6	50
PurchaseTickets	2	3	13	2	67
ReloadByCash	3	5	17	3	60
SetMonthlyPayment	2	78	77	28	36
ReloadFromAccount	3	7	34	3	43
TransferToAccount	2	7	16	2	29

4.4 Results Analysis

The result of the experiment indicates that the functional scenario-based test cases generation method is more effective if the test cases are generated based on relatively lower level specification. But the data in Table 2 shows that even the test cases are derived from the lowest level specification, some execution paths still cannot be tested. We think the ineffectiveness is caused by two reasons. The first reason is that the specification is not well defined, and the second reason is the refinement in program. These two reasons are also the causes of the relation "one scenario to multi paths" and "paths to no scenarios" mentioned in Section 3. The results confirms our analysis that the existence of these two relations usually reduces the rate of testing coverage.

Comparing to the unit testing, the situation faced by integration testing is more complex. The reason is that the test cases used in the testing are generated based on relatively higher level specification. Therefore, the ineffectiveness may be caused by the refinement in specification. In that case, the coverage of paths can be improved by considering the lower level specification in test cases generation process.

5 Proposals Based on Experience

In this section, we propose some suggestion based on our experience of using the test case generation method. Some proposals are proposed for improve the effectiveness of testing, and the others are proposed to make the process of test case generation more efficient and effective. We hope these suggestion can provide some helpful information to the testers who want to use the test case generation method.

Proposal 1. Let $\sim S_{pre} \wedge C_i \wedge D_i$ be a functional scenario in specification, extend set of test cases $G(\sim S_{pre} \wedge C_i)$ into $G((\sim S_{pre} \wedge C_i) \vee \neg(\sim S_{pre} \wedge C_i)) = G(\sim S_{pre} \wedge C_i) \cup G(\neg(\sim S_{pre} \wedge C_i))$

This proposal is used to generate test cases for the execution paths in relation to "paths to no scenario". We use *G(p)* to denotes the set of test cases derived from predicate *p*. The predicate $\neg(\sim S_{pre} \wedge C_i)$ in the proposal presents the functions undefined in the scenario $\sim S_{pre} \wedge C_i \wedge D_i$. The test cases derived from this predicate can be used to test the execution paths implementing functions that are not defined in the scenario. Note that this predicate can be constructed into a predicate expression in which the conjunction clauses may be the *testing condition* of other functional scenario in the same operation. And this will result in that the test cases derived from the predicate may satisfy other scenarios. Although it is possible to generate test cases from the same *testing condition* more than once, the duplication of generation do not affect the test coverage.

In order to test the paths in relation "one scenario to multi paths" that is caused by refinement in specification. The test cases must be generated from lower level specification. Usually, this situation happens in integration testing, since the test cases generated in integration testing is based on the higher level specification. Therefore the test cases generation process in integration testing should consider the refined specification, as reflected in **Proposal 2**.

Proposal 2. Let $F(x)$ be the disjunction of functional scenarios that contain input variable x. And let $F'(x)$ be the disjunction of functional scenarios which are in the decomposed module containing x. The test cases generated from the scenarios in higher level module should satisfy the condition: $\forall T'_c \in G(F'(x))$ $\cdot \exists T_c \in G(F(x))$ $\cdot T_c(x) = T'_c(x)$.

The notation T_c in **Proposal 2** denotes one test case of high level specification while the notation T'_c indicates one test case of lower level specification. $T_c(x)$ and $T'_c(x)$ present the value of input variable x in the test case T_c and T'_c respectively. **Proposal 2** does not provide a specific method that can be used in test case generation process. It is just a condition. If the generated test cases satisfy the predicate in **Proposal 2**, we can say that the lower level specification has been considered in the test case generation process.

Except for improving the effectiveness of testing, the following proposal is used to refine the process of generating test case. According to the original method, test case of one input variable will be generated based on one atomic predicate in testing condition first, and then check whether the generated test case satisfy the other atomic predicates in the testing condition. We find the selection of the atomic predicate used to generated test case effect the effectiveness of generating process pretty much. For example, assume that x is an input variable of type **nat0** (nature number), and predicate $x > 50 \wedge x < 100$ is a testing condition. There are two atomic predicates $x > 50$ and $x < 100$ in the test condition. Based on the original method, we first generate test case for x by adding Δx to 50. In practice, Δx is a number generated randomly. In this case, the value generated for x has infinite possibilities. So that the probability to generated value is smaller than 100 is very low. On the contrary, if we first generate test case for x based on atomic predicate $x < 100$. The range of the generated test case, [1, 99], is finite. The probability that the generated test case is larger than

50 is pretty high. Therefore, the predicate $x < 100$ should be selected to generate test case. We describe this idea generally in Proposal 3.

Proposal 3. Let $F(x)$ be the disjunction of testing condition that contain input variables x. Let $F(x) = ap_1(x) \wedge ap_2(x) \wedge \wedge ap_n(x)$, in which $ap_i(x)(i \in 1, 2, ..., n)$ indicates the atomic predicate consist of $F(x)$. The range of test cases of x that can generated based on $ap_i(x)$ is denoted as R_i, and the size of R_i is presented as $SZ(R_i)$. The atomic predicate ap_k, which is selected to generate test case should the property: $SZ(R_k) = min[SZ(R_1), SZ(R_2), , SZ(R_n)]$.

Applying **Proposal 3** in practice is very difficult. If the test cases are generated manually, the person who generates test cases can make the judgements based on his or her experience. But the original method is design to generate test cases automatically. To applying **Proposal 3**, a knowledge base or rule base system has to be built. Even though, selecting the appropriate atomic predicate is still difficult.

6 Raleted Work

Specification-based testing methods have been well researched based on different specification techniques. The test case generation method [1], which underlies our experiment is applicable to any operation specified in terms of pre- and post-conditions.The test case generation method based on algebraic specifications is introduced in [6], and the method of generating test case from reactive system specification is described in [7].

Cheon et al [8] use the assertions derived from formal specification in Object Constraint Language (OCL) as test oracles, and combine random tesing and OCL to carrying out automated testing for Java program. Michlmayr et al. introduce a framework of performing unit testing of publish/subscribe applications based on LTL specification in [9]. Khrushid et al. [**?**] present a framework named TestEra for testing Java program automatically based on specification. It employs Alloy analyzer for instance enumeration to generate all non-isomorphic test data [**?**]. Bandyopadhyay et al. [10] improve the existing test input generation method based on sequence diagrams of UML specification by consider the effects of the messages on the states of the participating objects.

Some approaches are proposed to enhance the effectiveness of the specification-based testing. Fraser et al. [11] investigate the effects of the test case length on the test result. Based on their experiments of specification based testing for reactive systems, they find a long test case can achieve higher coverage and fault detecting capability than a short one. They intend to improve the effectiveness of specification-based testing by change the length of test case. In [12], Liu et al. propose a technique called "Vibration" method to ensure all of the representative program paths of the program are traversed by the test cases generated from formal specification. This method provides a effective way for specification-based test case generation.

7 Conclusions and Future Work

In this paper, we performed two experiments to assess the functional scenario-based test cases generation method. Based on the static analysis and test results we find that this method is effective when the specification is well defined, but it may be ineffective if the specification is not well defined. We propose some improvement when the higher level specification are used in test case generation to ensure more execution paths can be tested. The final results show that our proposal can improve the effectiveness of the testing method. In the future, we intend to use a large-scale system to assess the test cases generation method and the proposals farther, and build a software tool to implement the testing method with our proposals.

References

1. Liu, S., Nakajima, S.: A Decompositional Approach to Automatic Test Case Generation Based on Formal Specification. In: Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement, pp. 147–155 (2010)
2. Li, C., Liu, S., Nakajima, S.: An Experiment for Assessment of a "Functional Scenario-based" Test Case Generation Method. In: Proceedings of International Conference on Software Engineering and Technology, pp. 64–71 (2012)
3. Liu, S.: Integrating Specification-Based Review and Testing for Detecting Errors in Programs. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 136–150. Springer, Heidelberg (2007)
4. Dick, J., Faivre, A.: Automating the Generation and Sequencing of Test Cases from Model-based Specifications. In: Larsen, P.G., Wing, J.M. (eds.) FME 1993. LNCS, vol. 670, pp. 268–284. Springer, Heidelberg (1993)
5. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer (2004) ISBN 3-540-20602-7
6. Gaudel, M.-C., Le Gall, P.: Testing Data Types Implementations from Algebraic Specifications. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 209–239. Springer, Heidelberg (2008)
7. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
8. Cheon, Y., Avila, C.: Automating Java Program Testing Using OCL and AspectJ. In: 7th International Conference on Information Technology, pp. 1020–1025 (2010)
9. Michlmayr, A., Fenkam, P., Dustdar, S.: Specification-Based Unit Testing of Publich/Subscribe Applications. In: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems Workshops, p. 34 (2006)
10. Bandyopadhyay, A., Ghosh, S.: Test Input Generation using UML Sequence and State Machines Models. In: International Conference on Software Testing Verification and Validation, pp. 121–130 (2009)
11. Fraser, G., Gargantini, A.: Experiments on the Test Case Length in Specification Based Test Case Generation. In: ICSE Workshop on Automation of Software Test, pp. 18–26 (2009)
12. Liu, S., Nakajima, S.: A "Vibration" Method for Automatically Generating Test Cases Based on Formal Specifications. In: 18th Asia-Pacific Software Engineering Conference, pp. 5–8 (2011)
13. `http://www.nta.go.jp/tetsuzuki/shinkoku/shotoku/tebiki2010/pdf/43.pdf`

Supporting Tool for Automatic Specification-Based Test Case Generation

Weihang Zhang[1,2] and Shaoying Liu[3]

[1] Graduate School of Software Engineering,
University of Science and Technology of China, Hefei, China
[2] Graduate School of Computer and Information Sciences,
Hosei University, Tokyo, Japan
Sea10494@mail.ustc.edu.cn
[3] Department of Computer and Information Sciences,
Hosei University, Tokyo, Japan
sliu@hosei.ac.jp

Abstract. Automatic test case generation is a potentially effective technique for program testing, but it still suffers from the lack of appropriate tool support. Our research presented in this paper mainly focuses on the developing of a tool for automatic test case generation based on formal specifications. We take advantage of the Liu's decompositional test case generation method and put forward a set of algorithms for automatically generating test cases based on various data types. A supporting tool on the application of the approaches is presented. The tool can generate test cases according to the users' given test conditions, and the result shows that our tool can produce test cases that satisfy most kinds of test conditions.

Keywords: automatic test case generation, specification, SOFL, decompositional method, functional scenario.

1 Introduction

Formal specification is one of the most important techniques of formal methods and it is used to precisely describe the most important information of the requirement for software systems. The target document of specification supported by our tool is the formal specification written in the SOFL, Structured Object-Oriented Formal Language [1]. It provides a practical method for developing software system and facilitating the subsequent development activities such as automatic test case generation and test result analysis.

Automatic test case generation based on formal specification is a potentially effective technique for software reliability. Several techniques are available for specification based test case generation. For instance, test case generated from algebraic specifications [2], from abstract state machines [3], and from B-method [4]. Liu et al. put forward a decompositional approach to automatic test case generation based on formal specifications [5]. The method is rigorous and practical, and it is good enough

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 12–25, 2013.

for realizing automation. However, there is no tool to support the entire automatic test case generation process. In this paper, we describe a supporting tool to support automatic test case generation based on SOFL specifications.

The structure of the supporting tool include generating test cases from various kinds of data types, such as Numeric Types, Character Types, String Types, Set Types, Sequence Types, and from compound predicate expressions, which include conjunction expressions and disjunction expression.

The remainder of this paper is organized as follows. Section 2 describes the concerned technique regarding the method of test case generation based on formal specification. Section 3 discusses the specific information on the design of the supporting tool. We will use a set of algorithms and some simple examples for illustration. Section 4 presents some details in a small experiment and introduces a prototype of the supporting tool. Section 5 introduces a brief overview of related work. Finally, we conclude the paper and point out future work in Section 6.

2 Approach to Automatic Specification-Based Test Case Generation

According to the work by Liu [5], the decompositional method of automatic test case generation based on formal specifications is concerned with generating a set of values that satisfy all the testing conditions. A testing condition of an operation specification is a constraint on the input variables and is expressed as predicate expression. With our method and the algorithms introduced in this paper, test case will be automatically derived from those predicate expressions.

In SOFL, the form of operation specification can be described as S(S_iv,S_ov)[S_pre,S_post], where S_iv denotes all input variables for the operation, S_ov represents all output variables whose values will be generated or updated after operation, and S_pre,S_post are the pre- and post-conditions of operation specification S, respectively.

1) Definitions: Suppose we have a post-condition of specification S, and it can be described as:

$$S_{post} = (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \ldots \vee (C_i \wedge D_i)$$

- Guard condition: A predicate Ci ($i \in \{1, 2 \ldots n\}$) is a "guard condition". The feature of guard condition is that it does not contain any output variables.
- Definition condition: A predicate Di($i \in \{1,2,\ldots n\}$) is a "definition condition", and there is at least one output variable but no guard condition.
- Functional scenario: In this case, a functional scenario $f_S$of S is a conjunction:Spre∧Ci∧Di.
- Functional scenario form (FSF): A disjunction expression $(S_{pre} \wedge C_1 \wedge D_1) \vee (S_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (S_{pre} \wedge C_i \wedge D_i)$ is a functional scenario form of specification S.

2) Testing condition: The testing condition in our method is the conjunction Spre∧Ci, where S_{pre} is the pre-condition, and C_i is the guard condition.

3) Test strategy: Suppose operation S has a FSF $(S_{pre} \wedge C_1 \wedge D_1) \vee (S_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (S_{pre} \wedge C_i \wedge D_i)$. Let T be a test set for S. Then, T must satisfy the condition $(\forall_{i \in \{1,2,..n\}} \exists_{t \in T} S_{pre}(t) \wedge C_i(t))$

The test strategy means that every testing condition must be tested and its corresponding test case should be found in the test set T.

3 Supporting Tool for Automatic Test Case Generation Method

In our work, we aim to produce a package in C# to support automatic test case generation from various kinds of predicate expressions based on the component-based software engineering approach. Before describing the supporting tool, we will explain how a test case can be derived.

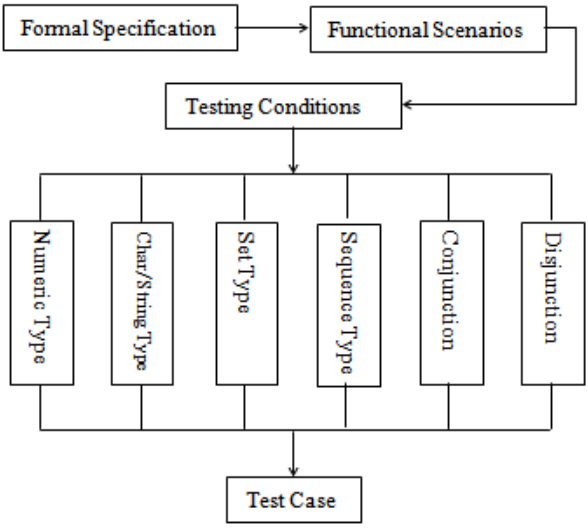

Fig. 1. Process of test case generation

Figure 1 shows the process of automatic test case generation based on the formal specification. As we have introduced in the previous sections, in order to generate test cases, functional scenarios derived from a formal specification must be given. And then, with the generated functional scenarios, we are able to obtain the testing conditions for the test. According to those derived test conditions, the supporting tool will be able to generate the corresponding test case in terms of different data types and expressions. In each chapter of this section, because the space of this paper is limited, the specific information about the data type and the introduction of their operators will be omitted. We just choose a few data types as examples and their corresponding algorithms, which can be used to generate test case based on different predicate expressions, for our discussion.

The algorithms for the operation of each data type are defined in each of the following classes, and all the classes are organized in the package named ASBTestCaseGeneration.

3.1 Test Case Generation Algorithms Based on Numeric Data Type

The algorithms are implemented using several methods in a class named Numeric. Each method deals with one specific case. The details of the methods are described below.

Method 1: GenerateFromSingleVar01 (P($x^{\sim}$): **string**, op: **string**) x: **real**{…}. The input variable in this method is $x^{\sim}$, and the output variable is x.

Algorithm 1: We first discuss the algorithm for simple predicate expressions involving only one input variable, and any predicate expression P($x^{\sim}$) can be transformed into the format as $x^{\sim}$ΘExp, where Θdenotes the relational operators of =, >, <, >=, <=, and <>, Exp is a constant expression which does not contain any variables. In such kind of situation, the algorithm for generating test cases is described below.

Suppose the predicate expression is expressed as the format of $x^{\sim}$ Θ Exp, then if Θ denotes =, we have x = Exp; And if Θ denotes >, then we have x = Exp + α, where α is a random positive numeric value. Also, if Θ denotes <, then we have x = Exp –α, and α is a random positive numeric value as well. For the other situations, such as Θ denotes >=, <=, <>, the methods for generating test cases are the same as above, and they will not be described in detail.

Method 2: GenerateFromSingleVar02 (P($x^{\sim}$): **string**, opt:**string**) x: **real**{…}.

Algorithm 2: Let us consider another format of simple predicate expression involving only one input variable, but the predicate expression P($x^{\sim}$) is organized as the format Exp1Θ Exp2, where Exp1 and Exp2 are both arithmetic expressions, and they may contain variable $x^{\sim}$.

For this situation, the algorithm for generating test cases is described below.

Suppose the predicate expression has the format Exp1Θ Exp2, we should transform Exp1Θ Exp2 into the format $x^{\sim}$ΘExp, and then apply the Algorithm 1 to generate the value of x. For instance, suppose we have a predicate expression 2x +5>x+1, then we can transform this expression into the format of x > -4. Finally, after applying the algorithm 1, we can generate the value of x with -4+α, where α is a random positive numeric value.

Method 3: GenerateFromMultiVar (P($x_1^{\sim}, x_2^{\sim}, \ldots, x_n^{\sim}$): **string**, opt: **string**) x_1, x_2, …, x_n: **real**{…}

Algorithm 3: The more complicated than the first two situations is when a predicate expression contains more than one input variables, and the predicate expression P($x_1^{\sim}, x_2^{\sim}, \ldots, x_n^{\sim}$)is expressed as the format Exp1Θ Exp2, where Exp1 and Exp2 are both arithmetic expressions, and they probably contain all the input variables

$x_1{\sim}, x_2{\sim}, \ldots, x_n{\sim}$.The algorithm to process such kind of expression will be described below.

In order to generate test cases to satisfy $P(x_1{\sim}, x_2{\sim}, \ldots, x_n{\sim})$,we should first make $x_1{\sim}$ as the variable to be discussed in our algorithm, then randomly generate appropriate values for the input variables $x_2{\sim}, x_3{\sim}, \ldots, x_n{\sim}$, respectively.

Eventually, we are able to derive the value of x_1 according to the method we have discussed for Algorithm 2.For example, suppose we have a predicate expression 3x+y+z > 2x +5. In order to automatically generate the values of x, y, z. Firstly, we should make x as the variable to be discussed in our method, and then randomly generate appropriate values for the input variables of y and z, such as y = 10, z = 20, therefore, the expression can be transformed into the format 3x+10+20 > 2x+5, which is suitable to apply the Algorithm 2 to generate the value of x. Finally, the test case satisfying $P(x_1{\sim}, x_2{\sim}, \ldots, x_n{\sim})$ is: x=-25+α, y=10, z=20, where α is a random positive numeric value.

Method 4: GenerateFromLinearExp ($P(x_1{\sim}, \ldots, x_n{\sim})$: **string**, opt: **string**) $x_1, x_2, \ldots, x_n$: **real**{…}

Algorithm 4: In order to effectively and efficiently generate test cases, we use a special data structure to operate test case generation for linear equation. The data structure is described below:

Index	Variable	Real

With such kind of data structure, any linear equation such as ax + b could be expressed as the form as:

a	x	b

For example, 3x - 5 can be expressed as

3	x	-5

Since every leaf node in binary tree can be transformed into the particular structure described above, we are able to calculate linear equation easily.

Suppose any linear relational expression can be expressed as: exp1Θ exp2, where exp1 and exp2 both represent arithmetic expressions, Θ denotes the relational operators of =, >, <, >=, <=, and <>. Suppose we have exp1 = ax + m, exp2 = bx + n. And the expression is exp1 = exp2. Then, let us make exp1 the form we described above as:

a	x	m

exp2 as:

b	x	n

After calculating, we have derived another expression px + q = 0 and it can be described as:

p	x	q

Where p = a - b, q = m – n.

Finally, we can generate the value of x according to the expression x Θ (–b) / a, where Θ is a relational operator.

For example, we try to generate a value of x from the expression 4x+5 = 2x-2, where exp1 = 4x+5, exp2= 2x-2, and Θ represents =. Then, we have the structure for exp1:

4	x	5

And structure for exp2:

2	x	-2

After calculating the arithmetic expressions p=4-2=2, q= 5-(-2)=7, we have another structure for the result:

2	x	-7

Eventually, a value we generated is x = (-7)/2 = -3.5

Method 5: GenerateFromQuaExp ($P(x_1{}^{2\sim}, x_2{}^{\sim}, \ldots, x_n{}^{\sim})$: **string**, opt: **string**) $x_1, x_2, \ldots, x_n$: **real**{…}

Algorithm 5: For quadratic equations:

As you can see, we are able to use this kind of structure to describe any kinds of

Index-a	Index-b	Index-c	Variable

quadratic equations such as ax^2+bx+c, and then we can get the corresponding structure as

a	b	c	x

For example, expression x^2+2x+4 can be described as the following form,

1	2	4	x

Since the method of transforming quadratic expression into the particular structure is similar to the linear expression, we can easily describe the specific structures for the quadratic equation of $x^2+2x-2=1$.

Then, the expression x^2+2x-2 can be described below,

1	2	-2	x

Accordingly, the value 1 will be described as

0	0	1	Null

After calculating, we have derived another expression $x^2+2x-3=0$. And it can be transformed into the structure as below,

1	2	-3	x

As we know, for quadratic equation, when $b^2 - 4ac \geq 0$, we can generate the values of x from the expression $x = \frac{-b \mp \sqrt{b^2-4ac}}{2a}$, and if $b^2 - 4ac < 0$, then we cannot get the value of x.

Here, we know that a=1, b=2, c=−3, and $2^2 - 4 * 1 * (-3) = 16 > 0$, so we can generate the values of x where $x_1 = \frac{-2+\sqrt{2^2-4*1*(-3)}}{2*1} = 1, x_2 = \frac{-2-\sqrt{2^2-4*1*(-3)}}{2*1} = -3.$

Finally, we can generate the test case from the quadratic equation that is $x_1 = 1$ and $x_2 = -3$.

Since the space of this paper is limited, we will not give the corresponding structure of binary tree in detail.

3.2 Test Case Generation Algorithms Based on Set Type

In this chapter, we focus our discussion on the algorithms of automatically deriving test cases from an expression involving all the input variables of the set type operator. Since the underlying principles of the algorithms for all the set type operators in SOFL are similar and the space of this paper is limited, we only choose some operators as examples for our discussion.

The algorithms are implemented using several methods in a class named Set. Each method deals with one specific case. The details of the methods are described below.

Method 6: GenerateFromSubset(x_1~: **set**)x: **set**, x_1: **set**{…}

Algorithm 6: Let us first use a simple example to explain the algorithm for the operator subset. Consider the predicate expression x subset x_1~. To generate a test case to satisfy this expression, according to the method introduced in Algorithm 3, we first randomly produce a set value for variable x_1~, and then in order to generate a test case, we just need to appropriately produce the values of x. We can take any elements in the generated set x_1 to make a new set value. Finally, the values of x_1 and x will satisfy the predicate expression, and they are the results of our test.

For example, suppose we want to generate a test case from the expression x subset x_1. Firstly, according the method, x_1 will be randomly generated, suppose it is {4,9,12}. And then, to decide the value of x, we just need to get some elements from the set x_1 we produced just now, suppose x is {9,12}. Finally, a test case for our test is x = {9, 12} and x_1={4,9,12}.

Additionally, for the expression x_1 union(x_2 inter x_3) subset x_4 uion x_5, where variables x_1, x_2,x_3,x_4,x_5 are all input variables of the set type, it is a compound expression involving different operators, and it will be discussed in subsequence sections.

Method 7: GenerateFromUnion(x_1~: **set**, x_2~: **set**)x: **set**, x_1: **set**, x_2: **set**{…}

Algorithm 7: Let us consider anther algorithm for the operator union. Suppose we have an expression x=union (x_1~, x_2~), wherex_1~, and x_2~ are all input variables of the operator union. To generate a test case for this predicate expression, we should also first randomly produce set values for variables x_1~, x_2~, and then it is quite simple to derive the result of the operation union (x_1, x_2). We can obtain all the elements of x_1 in the resulting set x and then add the members of x_2 that are not contained in x_1. Finally, the generated set values of x, x_1, and x_2 that satisfy the predicate expression are the test case for our test.

For example, suppose we have an expression x=union (x_1, x_2), to generate a test case from this expression, according to the algorithm, we should first randomly generate the values for the sets x_1 and x_2, suppose x_1 = {15, 17, 18, 20, 22} and x_2 = {8, 9, 17, 20, 23}. Then, obtain all the elements of x_1 to the set x, x = {15, 17, 18, 20, 22}. We can produce a suitable value for set x by adding the members of x_2 that

are not contained in x_1, so x will be {15, 17, 18, 20, 22, 8, 9, 23}. Finally, a test case for our test is x = {15, 17, 18, 20, 22, 8, 9, 23}, x_1 = {15, 17, 18, 20, 22} and x_2 = {8, 9, 17, 20, 23}.

Method8: GenerateFromInter(x_1˜: **set**, x_2˜: **set**)x: **set**, x_1: **set**, x_2: **set**{...}

Algorithm 8: Let us discuss the algorithm for the operator inter. Let x_1˜, x_2˜ are all input variables of the operator inter, and x=inter (x_1, x_2) is the target predicate expression. In order to generate a test case to satisfy the expression, the method is very similar to Algorithm 6 introduced above. Firstly, the set values for variablesx_1˜,x_2˜ will be randomly produced, and then we focus on how to generate values for variable x, we will give a pseudo code to explain this method:

```
Set Inter(Set s1, Set s2){
   Set result;
   for (i: =0 to s1.length - 1){
      k: = 0;
      while (s1 [i] != s2[k] && k< s2.length){
         k++;
      }
      if (k >= s2.length)
         i++;
      else{
         add s1 [i] to the set result;
         i++;
      }
   }
   return result;
}
```

Finally, we will obtain the result set value that represents the test case for variable x, and with the generated value of x_1, and x_2, we have successfully gained the test case for all input variables of the target predicate expression.

3.3 Test Case Generation Algorithms Based on Sequence Type

In this section, we will move forward to discuss the algorithms for automatically deriving test cases from a predicate expression involving all the input variables of the sequence type operator. As we mentioned above, because the underlying principles of algorithms for the operators in sequence type are quite similar, we just choose some operators (subsequence, elements and concatenation) as examples for our discussion, without giving all the descriptions for every operator in detail.

The methods for processing the sequence type are defined in a class named Sequence.

Mehtod 9: GenerateFromSubseq(S: **seq**, i: **int**,j: **int**) x: seq{…}

Algorithm 9: In this part, we will describe the algorithm for the operator subsequence.

Let S,i and j be input sequence variables, consider the predicate expression x = S (i, j), where i and j are both integer values, and the expression means obtaining the elements in sequence S from the position i to the position j, then make the obtained elements as a new sequence that is the subsequence of sequence S.

To generate a test case to satisfy this expression, according to the method introduced in Algorithm 3, we first randomly produce a sequence value for variable S, and the length of sequence S must be not less than j, and then in order to generate a test case for x, we just need to get values from the generated sequence S from the position i to the position j. The elements we got from sequence S will be added into the sequence of x. Finally, the values of S and x will satisfy the predicate expression, and they are the results of our test.

Method 10: GenerateFromElems(x_1: **seq**)x: set{…}

Algorithm 10: Let us discuss the algorithm for the operator elems. Let x_1, xbe the input and output variables of the operator elems, respectively. And x_1 is sequence type, x=elems(x_1) is the target predicate expression. To generate a test case for this predicate expression, we should also first randomly produce a sequence value for variables x_1 and then it is quite simple to derive the result of the operation x=elems(x_1). We can obtain all the elements from the sequence x_1, and then add the members to the set x to form a new set value. Finally, the generated set value of x, and sequence value of x_1 that satisfy the predicate expression are the test cases for our test.

Method 11: GenerateFromConc(x_1: **seq**,x_2: **seq**) x: seq{…}

Algorithm 11: Let us consider anther algorithm for the operator conc. Suppose we have an expression x=conc(x_1, x_2) , where x_1, and x_2 are all input variables of the operator conc. In order to generate a test case to satisfy the expression, the method is very similar to Algorithm 7 introduced above. The only difference is that in sequence, the duplication values are allowed to appear in a same sequence. Therefore, it is quite simple to generate test case for this operator. Firstly, we should randomly produce the sequence values for variables x_1, x_2, after that we include all the members of the generated sequence x_1 in the sequence x and then extend it by adding the members of the generated sequence x_2.

Finally, with the generated value of x, x_1 and x_2, we have successfully gained the test case for all input variables of the target predicate expression.

For example, in order to generate a test case from the expression x=conc(x_1, x_2), we should first randomly produce the values for variables x_1 and x_2, suppose x_1 = [1,2,3,4,5] and x_2 = [4,5,6,7,8]. Then, we are able to obtain the value for variable x by combining two sequence values of x_1 and x_2. Finally, x = [1, 2, 3, 4, 5, 4, 5, 6, 7, 8], x_1 = [1,2,3,4,5] and x_2 = [4,5,6,7,8] are the test cases for our test.

3.4 Algorithms for Automatic Test Case Generation Based on Conjunction and Disjunction Expressions

We have introduced each basic data type, and two compound data types of Set and Sequence in the previous sections. In this section, we will introduce the Conjunction expression and the Disjunction expression, respectively. In each compound predicate expression, no matter Conjunction or Disjunction, it will probably involve compound data types (e.g., numeric, string, set, sequence), which are introduced in the previous

sections. We will introduce the algorithms in detail on how to generate test cases according to those kinds of compound predicate expressions.

1) For Conjunction Predicate Expressions: We will describe the algorithm for conjunction expression in this section. The methods for dealing with the conjunction expression are defined in a class named Conjunction.

Method 12: GenerateFromConjunctionExp(exp: **string**) $x_1, x_2, \ldots, x_n$: **real** {…}

Algorithm 12: To generate a test case for conjunction, the test case must satisfy all the atomic predicate expressions in the conjunction. The fundamental idea for test case generation for a conjunction is that we should first generate a group of values for all the input variables of the operation from one of atomic predicate expression using the algorithms introduced in the previous sections. And then we test the values to make sure whether they satisfy other atomic predicate expressions or not, we will find a test case for the conjunction if the test case satisfy all the remaining conjunction constituents; Otherwise, it means the values are not suitable for the conjunction, and we should use the algorithm again to generate another test value, and repeat the above procedure until we find a suitable test case for all the constituents of the conjunction predicate expression.

In order to explain the main idea of the algorithm explicitly, a pseudo code will be given below:

```
voidGenerateFromConjunctionExp(String exp)
//The formal parameter exp is the target conjunction predicate expression, and the format of the expression is Q_i^1∧Q_i^2∧…∧Q_i^w. Each atomic expression and each operator in the atomic expression can be analyzed and detected, using the function ConstAnalyse(string exp); However, since the space in the paper is limited, the specific algorithm for this function will be omitted.
{
   j:= 1;
   successful:= true;
   ConstAnalyse(string exp) { … } //Using this function to analyze the expression, it will return a resulting  list which contains every separate atomic expression in the conjunction predicate expression Q_i^1∧Q_i^2∧…∧Q_i^w.
   GenerateFromAtomicExp(string aExp) // Generate r values v1,v2, … , vr as a test case that satisfies Q_i^j, aExp is a value from the list generated from function ConstAnalyse(string exp).
   j:=j+1;
   while(j<=w && j <const)   // const is a given number, in order to control the amount of Loop to avoid dead lock happening in the program.
   {
      if (Q_i^j (v1,v2,…,vr))
```

```
        j=j+1; // Q_i^j (v1,v2,…,vr)  means whether values
v1,v2, … ,vr satisfying the atomic expression Q_i^j or not, if
it returns true, it says that the derived values satisfy
the expression Q_i^j, otherwise, we have to generate the val-
ues again.
     else {
        successful = false;
        break;
     }
   }
   if (successful=true)
     Output the r values v1,v2,…, vr as a successful test
case;
   else
     Output a message "no test case is generated.";
}
```

2) For Disjunction Predicate Expressions: We will describe the algorithm for disjunction predicate expression. The methods for dealing with the disjunction expression are defined in a class named Disjunction.

Method 13: GenerateFromDisjunctionExp(exp: **string**) $x_1, x_2, \ldots, x_n$: **real** {…}

Algorithm 13: Compared with conjunction predicate expressions, test case generation from a disjunction seems much simpler. To generate test cases for the disjunction $Q_1 \vee Q_2 \vee \ldots \vee Q_m$, we just have to generate one test case for each disjunction constituent until all the atomic predicate expressions in the disjunction are covered, respectively. Finally, the generated test cases constitute a complete test set that is the result for the disjunction.

An algorithm of automatic test case generation from $Q_1 \vee Q_2 \vee \ldots \vee Q_m$ will be given below:

```
GenerateFromDisjunctionExp(String str) {
   j:= 1;
   ConstAnalyse(string exp) { … } //Using this function to
analyze the expression, it will return a resulting  list
which contains every separate atomic expression in the
conjunction predicate expression Q1∨Q2∨…∨Qm.
   while(j<=m)
   {
     Generate and output r values v1,v2,…,vr as a suc-
cessful test case that satisfies Qj using the algorithm
given in the previous sections.
     j:= j+1;
   }
   return;
}
```

4 Design of the Tool

In this section, we will briefly introduce the prototype tool for supporting the automatic specification-based test case generation methods. The support tool is implemented using Visual Studio .Net 2010 with language C#.

In order to explain our work clearly and help readers understand the techniques for the specification-base test case generation, we use a very simple case for illustration. According to the work done by Liu et al. [8], we assume that in our tool, the function of automatically generating all the test conditions from the derived functional scenarios of an operation, which are automatically generated from formal specification, have been realized. An example for the implementation of the tool will be given below.

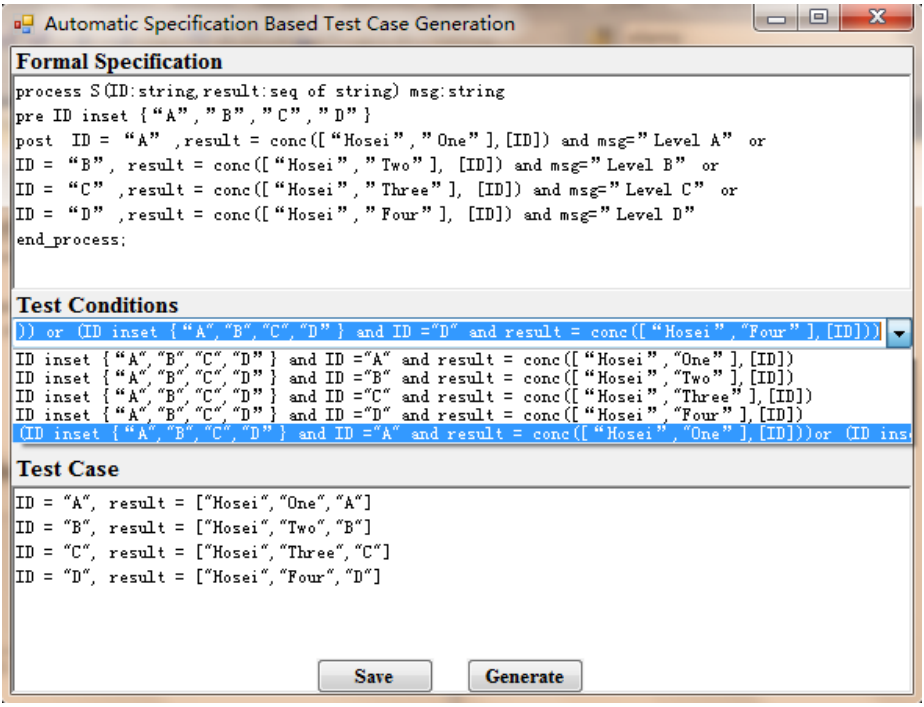

Fig. 2. Illustration of set type predicate expression

Figure 2 shows an illustration of processing compound predicate expressions, as we can see in the picture, test conditions for this process is ID inset {"A","B","C","D"} and (ID = "A" ,result = conc(["Hosei","One"],[ID])) or (ID = "B", result = conc(["Hosei","Two"], [ID])) or (ID = "C" ,result = conc(["Hosei","Three"], [ID])) or (ID = "D" ,result = conc(["Hosei","Four"], [ID])). The test conditions are associated with the Set and Sequence types, therefore, we should use the method for dealing with compound expressions. Eventually, the corresponding test case will be derived after processing.

5 Related Work

The decompositional approach to automatic test case generation based on formal specification was first introduced in Liu's paper [5], and it serves as a fundamental principle of the design of our supporting tool. Since the method just describe the main idea of automatic test case generation, in this paper, we have discussed some explicit algorithms. In additional, there have existed various methods for specification-based test case generation based on various specification techniques.

Bandyopadhyay et al.[3] put forward a testing methodology that combines information from UML sequence models and state machine models into one testable model based on the improvement of the work of Dinh-Trong et.al., which provided an approach to combine information from a class and a sequence diagram to generate test input. Based on state machine models, they use a testing method to select a set of transition sequences according to state machine coverage criteria, and then, with those generated transition sequences, the tool they built to support their approach is able to generate test inputs for each transition sequence.

Khrushid el al. [6] built a framework called TestEra for automatic testing of Java program based on specification. Their tool employs Alloy analyzer to produce instance of Alloy specification, where Alloy is a first-order declarative language based on sets and relations. After that, using the pre- condition of those generated instance of Alloy specification, the tool can automatically generate all non-isomorphic test inputs. Furthermore, TestEra can automatically generate the corresponding Jave data structure according to the description of the structural invariants of inputs.

Simon Burton [7] presents a framework of automatically generating tests for Z specification based on user-defined test criteria. Heuristics can be used to detect errors with the given resource constraints of the process. The framework allows for the automatic and formally generation of test sets based on formally defined testing heuristics. In the tool, test cases can be automatically generated by formalizing testing heuristics, analyzing properties of these heuristics.

6 Conclusion and Future Work

We have described the design and implementation of a supporting tool for automatic test case generation based on formal specifications. Formal specification in terms of pre- and post- conditions has tremendous advantages to be effectively utilized to generate test cases for testing programs. And tool support is crucial for the application of automatic test case generation approach based on formal specification. Our tool presented in this paper provides a package including many classes. Each class is designed to process each data type, respectively. Correspondingly, there are a lot of algorithms defined in each class for automatically generating test case according to different operators and predicate expressions. Our supporting tool is also crucial for the further research of automatic software testing. For example, our tool can serve as the foundation for testing result analysis, each component of this tool can be reused and integrated with the tool of testing result analysis easily.

In the future, we plan to make our further research to develop a set of more efficient algorithms for automatic test case generation. Since there are still some challenges in automatic testing, for example, it is difficult to deal with some set expressions including infinite set, such as x inset S, where S is a very large or infinite set. Therefore, in order to totally realize automatic testing, our future work should be focused on the algorithms that are capable to deal with all kinds of complicated expressions with a practically acceptable efficiency.

References

1. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer (2004) ISBN 3-540-20602-7
2. Gaudel, M.-C., Le Gall, P.: Testing Data Types Implementations from Algebraic Specifications. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 209–239. Springer, Heidelberg (2008)
3. Bandyopadhyay, A., Ghosh, S.: Test Input Generation using UML Sequence and State Machines Models. In: Proceedings of 2nd International Conference on Software Testing, Verification, and Validation (ICST), Denver, USA, April 1-4, pp. 121–130. IEEE CS Press (2009)
4. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
5. Liu, S., Nakajima, S.: Decompositional test case generation based-on specification. In: 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement, June 09-11 (2010)
6. Khurshid, S., Marinov, D.: TestEra: Specication-based Testing of Java Programs using SAT. Automated Software Engineering 11(4) (2004)
7. Burton, S.: Automated Testing from Z Specifications, TR YCS-2000-329, University of York, UK (2000)
8. Liu, S., Hayashi, T., Takahashi, K., Kimura, K., Nakayama, T., Nakajima, S.: Automatic Transformation from Formal Specifications to Functional Scenario Forms for Automatic Test Case Generation. In: 9th International Conference on Software Methodologies, Tools, and Techniques, Yokohama, Japan, September 29-October 1, pp. 383–397. IOS Press (2010)

A Formal Specification-Based Integration Testing Approach

Weikai Miao and Shaoying Liu

Department of Computer Science, Hosei University, Tokyo, Japan
weikai.miao.x1@stu.hosei.ac.jp, sliu@hosei.ac.jp

Abstract. It is well recognized that formal specification-based testing is a promising technique for software quality assurance. However, the application of this basic principle in integration testing is still facing the major challenge that most formal specification can precisely define the expected functions on system operations but fall short of offering a rigorous and intuitive representation of the system architecture that specifies the relations between the system operations, which leads to the difficulty in effective test data generation and test result analysis. In this paper we propose an integration testing approach based on the CDFDs (Condition Data Flow Diagram) of the SOFL (Structured Object-Oriented Formal Language) formal specification as a solution. Data flow paths are derived from the CDFDs. Test cases are then generated from the textual formal functional scenarios that precisely specify the expected functions on the system operations associated to the paths. The approach is described in detail by a running example. A case study is presented to demonstrate the feasibility and effectiveness of this approach.

1 Introduction

Formal specification-based testing is regarded as one of the most promising approaches to software faults detection [1][2]. Formal specification acts as a firm foundation for both test data generation and test results analysis, which can significantly enhance the quality of ultimated software systems. Main stream formal languages (e.g., Z [3], B [4] and VDM [5]) define system operations using formal notations that rigorously specify the initial and final states (e.g., pre- and post-conditions) of the operations. Both test cases and test oracles can then be easily generated from the formal definitions for testing the operations.

However, current formal specification-based testing methods face a challenge that most of them are powerful in unit testing but relatively difficult to be applied in integrating testing. One major reason is that most formal languages adopt mathematical notations for precisely defining the expected functions on system operations but fall short of intuitively describing the entire system architecture that specifies the interactions among the system operations. Subsequently, two problems need to be solved for effectively applying specification-based testing in integration testing. Firstly, inter-related pre- and post-conditions are not easy to

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 26–43, 2013.

be identified and organized for test data generation of integration testing, which leads to the difficulty in effective test case generation. The other problem lies in the test results analysis. Since textual formal specification does not explicitly describe interactions between system operations and system architecture, traditional path coverage approaches based on diagrams cannot be easily applied. Therefore, appropriate and intuitive test oracles can hardly be derived for the tester to analyze the test results.

As a solution to the above problems, conventional diagrams, for instance, data flow and control flow diagrams, are usually applied together with specific formal specifications for integration testing [6][7]. However, these diagrams just act as supplementary descriptions of textual formal specification; few of them are coherently integrated with textual formal specification due to the imprecise semantics of the graphical notations. As a result, formal specification-based integration testing are not widely accepted by most practitioners.

For effective integration testing, comprehensive formal models including both precise textual specifications and appropriate diagrams are demanded. To this end, in this paper, we propose an integration testing approach based on the CDFDs (Condition Data Flow Diagrams) of the *SOFL* (Structured Object-Oriented Formal Language) formal language [8][9][10]. In textual SOFL formal specification, each independent function of the system is formally represented by a SOFL process in terms of formal pre- and post-conditions. Relations among processes are rigorously described by the associated CDFDs intuitively.

The main principle underlying our integration testing approach is to first derive all potential paths from the CDFDs and then generate adequate test cases from the involved processes of these paths. Specifically, the pre- and post-conditions of each process are further transformed into a formal functional scenario form that is a disjunction of functional scenarios. Each functional scenario rigorously specifies an independent execution of the system operation associated to the process. All the involved functional scenarios of a path constitute a set of functional scenario sequences that are covered by adequate test cases. Therefore, intuitive diagrams can be used for rigorous paths extraction and test result analysis, and test cases can also be generated from the textual formal specification. The approach strikes a balance between rigorous testing and intuitive test result analysis, which is easy to be applied by practitioners.

The rest of this paper is organized as follows. A brief introduction to the SOFL language is presented in Section 2. Techniques of the integration testing approach are presented in Section 3. Section 4 describes the case study. Section 5 gives the comparison with the related work. Finally, we conclude the paper and point out future research directions in Section 6.

2 A Brief Introduction to the SOFL Language

Before going into the technical details of the integration testing approach, it is necessary to briefly introduce the SOFL.

As a formal specification language, SOFL is designed by integrating different notations for constructing a precise and comprehensible formal specification.

A SOFL formal specification consists of a group of *modules* that are organized in a hierarchical manner. Each module encapsulates the related *processes* that specify the expected functions, the *data stores* that specify the data resources accessed by the processes, and the *invariants* that specify the constraints to be conformed by the processes and data stores. Each process here is the basic specification component for defining the expected functions. A process mainly consists of the input ports and the output ports for describing the input and output variables, and the pre- and post conditions specifying the semantics.

Distinguished from other formal specification languages, the SOFL formal language offers the CDFDs (Condition Data Flow Diagram) that integrates conventional Data Flow Diagrams and VDM-SL [5] in a coherent manner to intuitively, rigorously and comprehensibly describe the system architecture.

Figure 1 describes a typical SOFL specification with its corresponding CDFD.

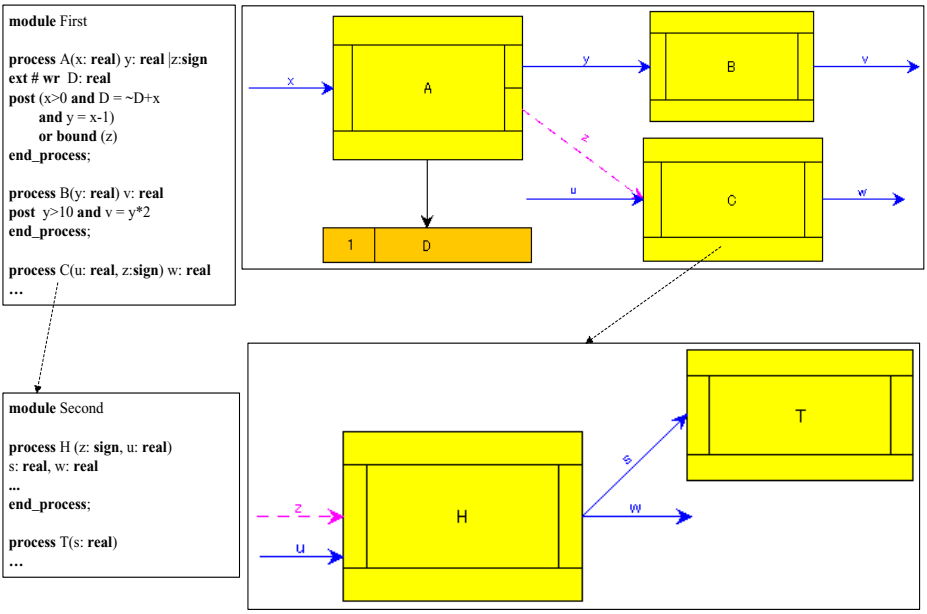

Fig. 1. An Example of the SOFL Formal Specification

The left part of Figure 1 describes a SOFL textual formal specification of two *modules*. The right part of this figure describes the corresponding CDFDs. Module $First$ consists of processes A, B and C. These three processes are connected by data flows, which represents the overall function of module $First$.

A process is composed of five parts: *name*, *input port*, *output port*, *pre-condition* and *post-condition*. The name of a process is an identifier. The input and output ports specify the input and output variables of the process. The pre- and post-conditions rigorously define the semantics of the process. The semantics of

a process is interpreted as follows: when one of the input ports is available, which means that all of its input variables are bound to specific values in their types, the process will be executed. As a result of the execution, one of the output ports is made available, which means that all of its output variables are bound to specific values of their types. If the input variables satisfy the pre-condition before the execution, the output variables are required to satisfy the post-condition after the execution of the process, provided that the execution terminates.

For instance, process A consists of one input port and two output ports. It takes x of real type as the input variable and produces either y or z as the output variable. The pre-condition of process A is set to be *true* which can be omitted; the post-condition requires that the output variable y is equal to $x - 1$ if x is greater than 0, and the external variable D will be updated by following condition $D = \tilde{}D + x$; otherwise variable z will be made available.

In each CDFD, a process is represented by a rectangle box with a name in the center. Each input port is denoted by a narrow rectangle on the left part of the process box, which receives input data flows. Similarly, each output port is denoted by a narrow rectangle on the right part of the process, which produces output data flows. The pre- and post conditions are denoted by rectangles located in the upper and lower parts of the process. For example, process B receives a data flow y given by process A and produces an output data flow v.

Modules are organized into hierarchical structures to represent the system architecture. In this example, process C is decomposed into a lower-level module *Second* which consists of processes H and T.

3 CDFD-Based Integration Testing Approach

Following the SOFL method, formal specification is constructed and affiliated with hierachical CDFDs for describing the system architecture. The bottom-up strategy for integration testing can be adopted since lower-level CDFDs are abstracted as processes in the higher-level CDFDs. Specifically, the integration testing starts from the bottom level system modules. When a lower-level module is thoroughly tested, it is nested as a process of its higher level system module for integration testing. This iterative procedure continues until the top-level system module is finally tested. Therefore, the fundamental problem to be resolved is the integration testing of each individual module described by a CDFD. The following steps can be taken for the integration testing of each module.

1. extracting all independent paths from the CDFD;
2. transforming the pre- and post-conditions of each process involved in each path into the *functional scenario form*;
3. constructing the *functional scenario sequences* of the involved processes of each path, and then testing the system using the test cases generated from the constructed functional scenario sequences;

Figure 2 shows the CDFD of the running example that will be used for illustrating our approach. The corresponding formal specification is described in Figure 3.

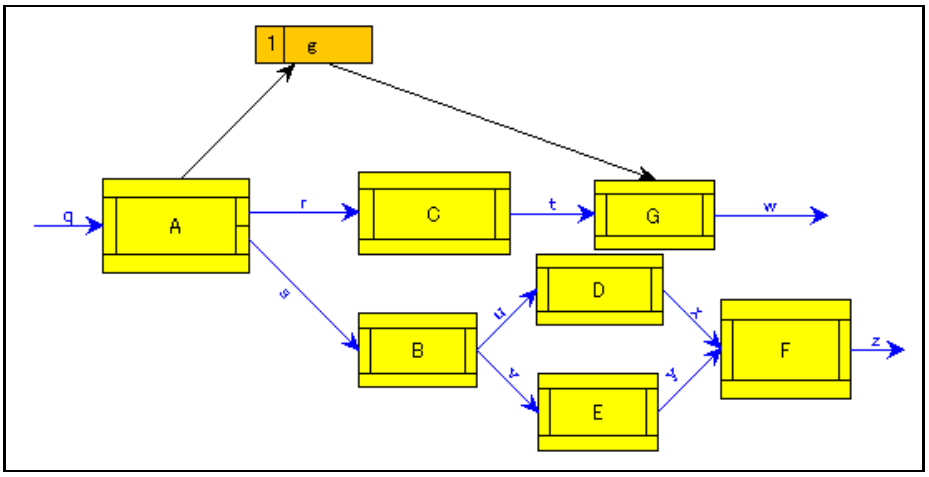

Fig. 2. CDFD of the Running Example

3.1 Path Extraction from CDFD

A CDFD specifies the interactions among a set of processes using data flows.

Definition 1. *Let D be a universal set of data flows and M be a universal set of SOFL processes involved in a CDFD, a path p of the CDFD is defined in the format $p \equiv (r_1, r_2, ..., r_n)$ where $r \in D \cup M$.*

A path is an ordered sequence of processes connected by data flows, starting from a starting process and ending at a terminating process. A starting process is a process whose input data flows are not the output data flows of any other process in the same CDFD. A terminating process, on the contrary, is a process whose output data flows are not the input data flows of any other process in the CDFD. Based on the definition, data flow paths can be directly extracted from the CDFD. Note that the control flow is treated as a special data flow in the context of this paper.

In the CDFD shown in Figure 2, starting node process A forks two independent data flows r and s from two output ports, respectively. Processes G and F are two terminating processes, since none of their output data flows is received by any other process. Therefore, two paths can be extracted from this CDFD. The first path is (q, A, r, C, t, G, w) in which q, r, t and w are data flows. Process A is the starting process of the path, and process G is the terminating process.

The other path is $(q, A, s, B, u|v, D|E, x|y, F, z)$ which consists of processes A, B, D, E, and F, and the related data flows q, s, u, v, x, y, and z. Starting process A receives data flow q and produces the output s to process B. Processes D and E receive two data flows u and v in parallel from process B; similarly, process F receives data flows x and y in parallel as its input data.

```
process A (q: nat0) r: nat0 | s: nat
ext wr g
post q < 0  and  s = q*5
     or  q = 0  and  r = 0  and  g = ~g*r
     or  q > 0  and r = q^2  and g = ~g+r
end_process;

process C (r: nat0) t: nat0
pre r >= 0
post r > 0  and  r < 64  and  t = r*4
      or  r > 64  and  t = r-64
      or  r = 0  and  t = r+1
end_process;

process G (t: nat0) w: nat0
ext rd g
post t < 35  and  w = ~g*2-t
     or  t >= 35  and  w =~g+t
end_process;
```

Fig. 3. Formal Specification of the Running Example

3.2 Transformation of Functional Scenario Form

Each path extracted from the CDFD is a set of processes connected by data flows, which will be covered by adequate test cases. To facilitate the test case generation, the pre- and post-conditions of each process involved in each path is transformed into an equivalent disjunction of *functional scenarios*, each describing an independent function in terms of the input and output relation [10].

Specifically, let $P(inP, outP)[preP, postP]$ denotes a formal process, where inP is the set of all the input variables; $outP$ is the set of all the output variables. $preP$ and $postP$ denote the pre- and post conditions of P. The pre- and post-conditions of P precisely define its functional behaviors in terms of the relations between the input and output variables.

Definition 2. *Given a formal process P in terms of pre- and post-conditions, let the post-condition $postP \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee ... \vee (C_n \wedge D_n)$, where each C_i $(i = 1, ..., n)$ is a predicate called a guard condition that contains no output variable and D_i a defining condition that contains at least one output variable but no guard condition. Then each $\tilde{}preP \wedge C_i \wedge D_i$ $(i = 1, ..., n)$ is called a functional scenario.*

A functional scenario $preP \wedge C_i \wedge D_i$ describes that if the pre-condition $preP$ and guard condition C_i are both true, the output of the process is defined by defining condition D_i. All of the functional scenarios of the process are expected to cover all the behavioral situations when the input satisfies the pre-condition. To test whether the associated system operations implement the required functions

specified by process P, test cases need to be generated from each conjunction $\tilde{}preP \wedge C_i$.

Definition 3. *Given a formal process* $P \equiv (\tilde{}preP \wedge C_1 \wedge D_1) \vee (\tilde{}preP \wedge C_2 \wedge D_2) \vee ... \vee (\tilde{}preP \wedge C_n \wedge D_n)$*, each conjunction* $\tilde{}preP \wedge C_i$ $(i = 1, ..., n)$ *is called the* testing condition *of the scenario* $\tilde{}preP \wedge C_i \wedge D_i$.

For a test case t generated from the testing condition $\tilde{}preP \wedge C_i$, if the real execution result r does not satisfy the defining condition D_i, then we assert that the system operations do not correctly implement the functional scenario $\tilde{}preP \wedge C_i \wedge D_i$. To sufficiently test the associated operations of a process, test cases are required to cover all functional scenarios of the process.

The merits of the functional scenario forms lie in two aspects. Compared with conventional disjunctive normal forms of pre- and post conditions, functional scenario form is more appropriate in describing sophisticated semantics. Moreover, precise test oracles can be easily derived based on the relatively readable functional scenario forms so that an effective test analysis can be achieved.

3.3 Test Case Generation Based on Functional Scenario Sequence

Since a path is a sequence of processes connected by data flows, the overall functions represented by a path are actually specified by different functional scenario sequences contributed by the involved processes; each process along the path contributes one functional scenario for each functional scenario sequence.

Figure 4 describes one path derived from the CDFD of our running example. The extracted functional scenarios of the processes involved in the path are listed in Figure 5.

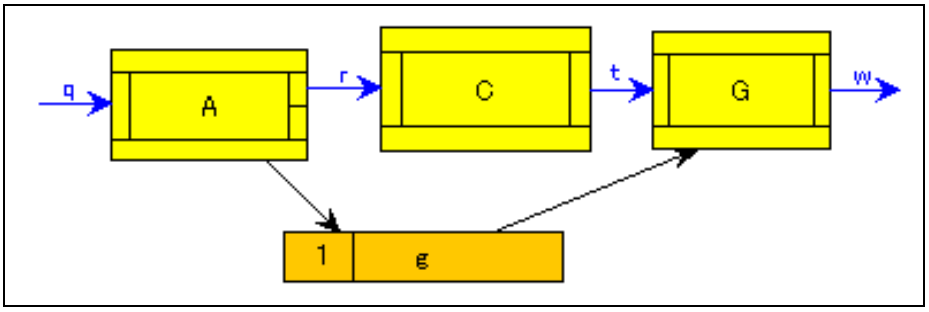

Fig. 4. A Path of the Running Example

The path (q, A, r, C, t, G, w) is represented by the sequence of the three involved processes A, C, and G, and the data flows connecting them. For simplicity, we use f_{ni} to denote the ith functional scenario of process n. For example, f_{A2} is the second functional scenario of process A. As shown in Figure 5, the

f_{A1}: q > 0 **and** r = q^2 **and** g = ~g+r	f_{C2}: r > 64 **and** t = r-64
f_{A2}: q = 0 **and** r = 0 **and** g = ~g*r	f_{C3}: r = 0 **and** t = r+1
f_{A3}: q < 0 **and** s = q*5	f_{G1}: t < 35 **and** w = ~g*2-t
f_{C1}: r > 0 **and** r < 64 **and** t = r*4	f_{G2}: t >= 35 **and** w = ~g+t

Fig. 5. Functional Scenarios of the Processes Involved in the Path of the Running Example

testing condition of f_{A2} is $q = 0$ which specifies the constraints on the input variable q of process A. The defining condition of f_{A2} is $r = 0$ **and** $g = \tilde{}g * r$ which precisely describes how the output variable r and the data store g are defined.

Suppose a path is composed of n processes ranging from P_1 to P_n, the number N of the possible functional scenario sequences of the path is equal to $\prod_{i=1}^{n} F(P_i)$ where function F returns the number of functional scenarios of a process P_i. To sufficiently test a path, an intuitive way is to generate test data from all the $\prod_{i=1}^{n} F(P_i)$ functional scenario sequences.

Processes A and C provide three functional scenarios, respectively; and process G offers two functional scenarios. Therefore, 18 $(3 * 3 * 2 = 18)$ functional scenario sequences can be derived from the involved processes of this path. Each functional scenario sequence consists of three functional scenarios provided by the three involved processes of the path, respectively. For instance, sequence (f_{A1}, f_{C1}, f_{G1}) is one of the derived functional scenario sequences of the path.

However, not all functional scenario sequences can be used for test cases generation. For some functional scenario sequences, the outputs specified by the defining condition of a predecessor functional scenario never satisfy the testing condition of the successor functional scenario. Thus, no test data can be derived from the entire functional scenario sequence. For instance, functional scenario sequence (f_{A2}, f_{C1}, f_{G1}) of the above example cannot be used for test data generation since no output value of variable r defined by the defining condition $r = 0$ of f_{A2} can satisfy the testing condition $r > 0$ of f_{C1}. Therefore, no test data can be generated from the functional scenario sequence. Only feasible functional scenario sequence can be used as the foundation for test case generation.

Definition 4. *A functional scenario sequence* $(f_1, f_2, ..., f_n)$ *is feasible* if and only if *there exists at least one initial input that leads to the satisfactory of each testing condition of each involved functional scenario* f_i $(i : 1, ..., n)$.

In our example, (f_{A1}, f_{C1}, f_{G1}) is a feasible functional scenario sequence. Given input value 2 of the input variable q of the first functional scenario f_{A1}, the testing condition $q > 0$ of f_{A1} is satisfied. Suppose data store variable $\tilde{}g$ is 0, the output variable r is equal to 4 and g is updated to be 4. In this case, the

second functional scenario f_{C1} of the functional scenario sequence is activated since its testing condition $r > 0$ **and** $r < 64$ is satisfied by the value 4 of r. Thus, the value of its output variable t is evaluated to be 16. Subsequently, the last functional scenario f_{G1} is activated since its testing condition $t < 35$ is satisfied by the value of t. The final output value of variable w should be equal to -8.

Based on the feasible functional scenario sequences derived from each path, test cases can be generated. It is well known that providing the correctness assurance by testing is infeasible in practice. What we can do is to generate adequate test cases that allow every functional scenario sequence to be exercised at least once (the more, the better), which is reflected in the following criterion.

Criterion 1. *Let* $H \equiv (preP_1 \wedge C_1 \wedge D_1, preP_2 \wedge C_2 \wedge D_2, ..., preP_n \wedge C_n \wedge D_n)$, $(n \geq 1)$ *be an functional scenario sequence of a path* p *where* $preP_i \wedge C_i \wedge D_i$ *denotes a functional scenario of an involved process of* p. *Let* T *be a test set (a set of test cases). Then,* T *is said to satisfy the* scenario coverage *of* H iff *for each test data* $t \in T$, t *satisfies the conjunction of all the testing conditions* $\bigwedge_{i=1}^{n} preP_i \wedge C_i$ *of the* n *processes of* H.

In order to generate a test set T that specifies the required condition in this criterion, we can generate at least one test case to cover the functional scenario sequence H. However, directly generating test data from complete functional scenario sequences of the data flow paths is not easy to be achieved and also inefficient for detecting errors at early stages of testing.

A more practical way to test case generation is to take the incremental strategy. For each path under test, each feasible functional scenario sequence is dynamically constructed for deriving test cases. Specifically, the first step is to select one functional scenario of the starting process of the path. After running the corresponding test cases generated from the functional scenario, one functional scenario of the second process is then selected and combined with the functional scenario of the starting process for constructing a functional scenario sequence. Then the feasibility of the current functional scenario sequence is checked according to Definition 4. If the functional scenario sequence is infeasible, another functional scenario of the second process is selected. Otherwise, one functional scenario of the third process along the path is added into the functional scenario sequence. This procedure continues until one functional scenario of the terminating process along the path has been added into the functional scenario sequence. Once one functional scenario of the terminating process is added a complete feasible functional scenario sequence is constructed for covering the entire data flow path. A path is sufficiently tested if and only if all of its functional scenario sequences are covered by test cases.

To check whether the system correctly implements the expected functions specified by the functional scenario sequences, a test oracle is constructed.

Let $G_n \equiv \bigwedge_{i=1}^{n} preP_i \wedge C_i$ be the conjunction of all the testing conditions of the processes of H, and t be a test case that covers G_n. Let q_i be the inputs and r_i be the execution result of the ith functional scenario of H by running the software

under test using t, respectively. r_i is expected to satisfy the defining condition D_i of the ith functional scenario $preP_i \wedge C_i \wedge D_i$ of H, but if this is not true, it will imply a difference between the behavior required by the functional scenario sequence and the behavior provided by the software. Formally, the condition

Condition 1. $R(H, t, r_i) \equiv G_n(t) \wedge \bigvee_{i=1}^{n} \neg D_i(q_i, r_i)$

is used as a test oracle to determine whether such a difference exists. If this condition holds, we assert that the system under test does not correctly implement the expected functions.

The test case generation and the test result analysis procedures presented above are implemented by the following algorithm in JAVA pseudo code. In this algorithm, **Num** is a function that returns the number of functional scenarios of a process. Function **Conc** is a *concatenation* operator of sequences. Function f_q represents the qth functional scenario of a process, and function R implements the test oracle of condition 1.

Algorithm 1. boolean IntTest (**p***: an n-length array representing a path of n processes*){
  A*: an empty set of functional scenario sequence;*
  $k = \mathbf{Num}(\mathbf{p}[1])$;
  for$(i = 1, ...k)$ $\mathbf{A} = \mathbf{A} \cup \{f_i(\mathbf{p}[1])\}$
  for$(j = 2, ..., n)$**{**
    $k = \mathbf{Num}(\mathbf{p}[j])$;
    for *each element* $a \in A$ **{**
      for$(q : 1, .., k)$ **{**
       $a = \mathbf{Conc}(a, (f_q(\mathbf{p}[j])))$;
        if$(a$ *is infeasible*$)$ $\mathbf{A} = \mathbf{A}/\{a\}$;
      }
    }
    for *each element* $a \in A$ **{**
      Generating a test set T for a;
      if $(\exists_{t \in T} \cdot R(a, t, r))$ **return** $false$*;*
    }
  return *true;*
  }

Following the algorithm, functional scenarios of all the processes along a path are gradually picked up for constructing the functional scenario sequences. The set of functional scenario sequence **A** is initially empty. In this paper, we assume that the functional scenarios of each starting process are covered by adequate test cases at the stage of unit testing. Therefore, all the functional scenarios of the starting process involved in the path are directly added into set **A**.

When dealing with each process $\mathbf{p}[j]$, each functional scenario sequence a in set **A** will be be concatenated with each functional scenario of $\mathbf{p}[j]$, respectively. Then the feasibility of each functional scenario in the updated set **A** is checked,

and infeasible functional scenario sequences are removed from **A**. For each functional scenario sequence a in set **A**, a test set T is generated and the test result r obtained by running each test case t ($t \in T$) will be analyzed in the context of the test oracle defined by Condition 1. If any software fault is detected, the algorithm is terminated and the $false$ signal is responded. When all the functional scenarios involved in **A** are covered by test data, functional scenarios of the next process along the path are added into **A**. This procedure continues until all the feasible functional scenario sequences of a path are covered by test cases.

Table 1 describes a part of the testing procedure for covering the path (q, A, r, C, t, G, w) shown in Figure 4.

Table 1. Test Case Generation of the Running Example

current functional scenario sequences	**test case**	**test result**	**expected result**
(f_{A1}, f_{C1})	$q = 7$	$t = 196$	$t = 196$
(f_{A1}, f_{C2})	$q = 9$	$t = 17$	$t = 17$
(f_{A1}, f_{C3})	/	/	/
...	...	...	...
(f_{A1}, f_{C1}, f_{G1})	$q = 1$	$w = -2$	$w = -2$
(f_{A1}, f_{C1}, f_{G2})	$q = 6$	$w = 144$	$w = 180$

As the first step, all functional scenarios of the starting process A are added into the set of functional scenario sequences. Then each functional scenario of the second process C is added into the set and concatenated with the functional scenarios of process A. To cover the current functional scenario sequence (f_{A1}, f_{C1}), we generate the test data 7 for the input variable q of f_{A1} to run the system. As the result of executing the test data, the value of the output variable t provided by the system under test is equal to 196, which satisfies our expected output value. Similarly, (f_{A1}, f_{C2}) is also covered by a test case.

However, we find that functional scenario sequence (f_{A1}, f_{C3}) is infeasible according to Definition 4. Therefore, this functional scenario sequence is eliminated. When all the functional scenario sequences contributed by the first and second processes are covered by test cases, the terminating process G of this path is added after process C for integration testing. As an example, (f_{A1}, f_{C1}, f_{G1}) is required to be covered by test cases. We assume that the current value of variable g is 0. After running test data 1 of input variable q, the real output value of variable w is -2, which satisfies the expected output results.

Then we proceed to generate test data for the functional scenario sequence (f_{A1}, f_{C1}, f_{G2}). We assume that the current value of variable g is set to 0. After running test data 6 of input variable q, the real output value of variable w is 144, which violates the expected output value 180 inferred from f_{G2}. Therefore, the test oracle is evaluated to be true, which indicates that the functional behavior of the system under test is inconsistent with the functional scenario sequence. In this case, we can determine that some faults exist in the system under test.

4 A Case Study

We have conducted a case study for evaluating the effectiveness of our approach.

4.1 Background

The software of our case study is an *Online Travel Agency System* (*OTAS*), which offers major functions for travelers including *tickets reservation operations*, *hotel reservation operations*, *payment operations*, and etc. Since the system is too large to be described within this section, we just choose one sub-system called *Online Hotel Reservation Sub-system* (*OHRS*) of the *OTAS* for evaluating our testing approach. CDFDs of the *OHRS* are shown in Figure 6.

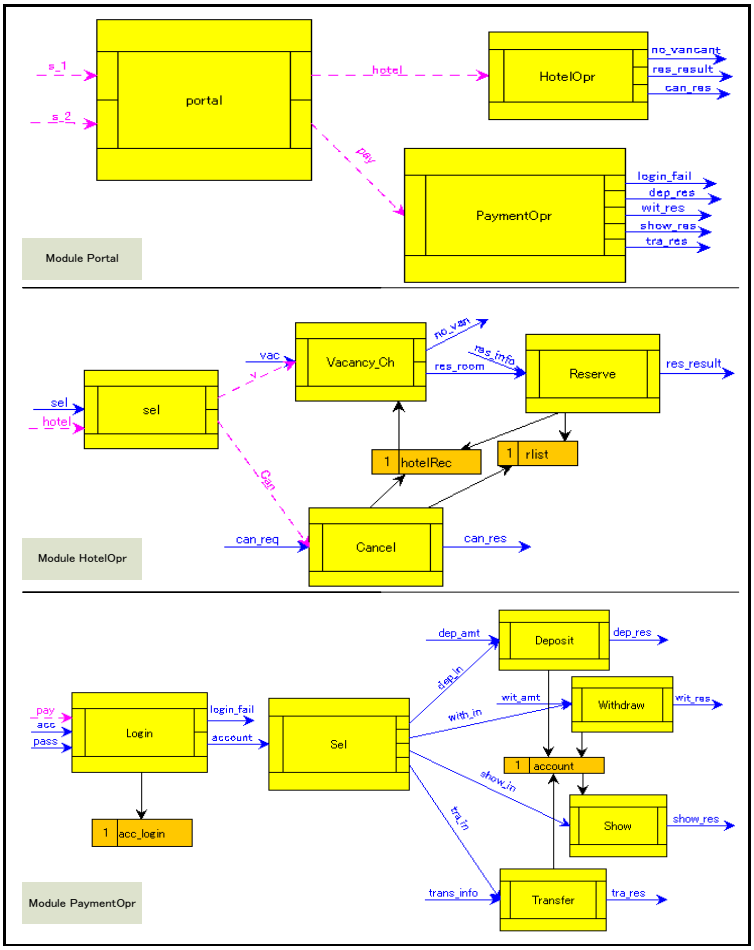

Fig. 6. CDFD of the *OHRS*

The top-level module *Portal* of *OHRS* consists of three processes *portal*, *HotelOpr* and *PaymentOpr*, which is described by the CDFD shown in the top part of Figure 6. In this module *Portal*, process *portal* takes control flows *s_1* and *s_2* as input data, and produces output control flows *hotel* and *pay* for activating process *HotelOpr* and *PaymentOpr* respectively.

Processes *HotelOpr* and *PaymentOpr* are actually lower-level modules that abstract the functions of hotel and payment operations. These two modules are described by the CDFDs shown in the middle and lower parts of Figure 6.

In the CDFD of module *HotelOpr*, process *sel* receives requests for reserving rooms or cancelling reservation. Process *Vacancy_Ch* stands for the function of vacant room checking and process *Reserve* represents the function of room reservation. Process *Cancel* is responsible for handling reservation cancellation. Two data stores *hotelRec* and *rlist* represent the necessary data records of room and reservation information.

In the CDFD of module *PaymentOpr*, process *Login* represents the function of account authentication. Process *sel* offers options for different operations including depositing and withdrawing currency, showing balance and transferring currency between accounts. These operations are specified by processes *Deposit*, *Withdraw*, *Show* and *Transfer*. Data stores *acc_login* and *account* represent valid user information and detailed account information.

In order to evaluate the ability in software error detection of integration testing approach, we insert 58 errors into the program of the *OHRS*. The program is rigorously implemented according to the specification and CDFDs. Before inserting the software errors, we also have checked the conformance of the operations involved in the program to their corresponding formal processes in the specification via unit testing.

4.2 Results and Analysis of the Case Study

Eight paths are extracted from the CDFDs of our *OHRS*, which are listed in Figure 7.

For example, *path 2* stands for the execution scenario of hotel room reservation, either a successful or a failed reservation. The functions represented by *path 2* are precisely specified by the involved functional scenarios of the three processes. Specifically, process *sel* produces one functional scenario of selecting reservation operation. Process *Vacancy_Ch* produces the functional scenario of a successful vacant room checking. Process *Reserve* produces two functional scenarios that represent the successful and failed reservation, respectively. Therefore, two functional scenario sequences ($2 = 1 * 1 * 2$) are derived from *path 2*, which represent the successful and the failed room reservation, respectively. For a concise illustration, we just focus on the functional scenario sequence of successful room reservation. The corresponding involved functional scenarios of the functional scenario sequence are described in Figure 8.

As shown in Figure 8, the functional scenario of process *sel* represents that if the signal variable *hotel* is available and the input variable *sel* is equal to 0, the signal variable *v* will be available. The functional scenario of process

Paths

1. ([sel, hotel], P_{sel}, [v, vac], [$P_{Vacancy\_Ch}$, hotelRec], no_van)
2. ([sel, hotel], P_{sel}, [v, vac], [$P_{Vacancy\_Ch}$, hotelRec], [res_room, Res_info], [$P_{Reserve}$, hotelRec, rlist], res_result)
3. ([sel, hotel], P_{sel}, [Can, can_req], [P_{cancel}, hotelRec, rlist], can_res)

4. ([pay, acc, pass], [P_{Login}, acc_login], login_fail)
5. ([pay, acc, pass], [P_{login}, acc_login], account, P_{Sel}, [dep_in, Dep_amt], [$P_{Deposit}$, account], dep_res)
6. ([pay, acc, pass], [P_{Login}, acc_login], account, P_{Sel}, [with_in, wit_amt], [$P_{Withdraw}$, account], wit_res)
7. ([pay, acc, pass], [P_{Login}, acc_login], account, P_{Sel}, Show_in, [P_{show}, account], show_res)
8. ([pay, acc, pass], [P_{Login}, acc_login], account, P_{Sel}, [tra_in, trans_info], [$P_{Transfer}$, account], tra_res)

Fig. 7. Paths Derived from the CDFDs of *OHRS*

```
process sel (sel: nat0, hotel: sign) v: sign |Can: sign
post      sel = 0 and bound (hotel) and bound (v)
          or
          ...
end_process

process Vacancy_Ch (v: sign, Vac: Check_Vacancy) res_room: ResRequest | no_van: string
ext # rd hotelRec
post   bound (v) and ( exist [x: hotelRec] | x.stay_period = vac.stay_period and x.amount >= vac.amount )
       and res_room.stay_period = vac.period and res_room.amount = vac.amount
       or
       ...
end_process

process Reserve (res_room: ResRequest, res_info: ResInfo) res_result: string
ext # wr hotelRec
    # wr rlist
post  (exist [x: hotelRec] | x.stay_period= res_room.stay_period and x.amount >= res_room.amount )
            ...
        or
      …
end_process
```

Fig. 8. Involved Functional Scenarios of *Path 2*

1. sel = <0>, stay_period = <20110901-1>, amount = <1>, res_info.name = <"Nomura">, res_info.email = <"nomura@...">

2. sel = <0>, stay_period = <20110901-3>, amount = <2>, res_info.name = <"Mitsui">, res_info.email = <"Mitsui@...">

…

Fig. 9. Sample Test Cases for Testing the *OHRS*

Vacancy_Ch represents that if the input signal variable v is available and if there exist enough rooms recorded in data store *hotelRec* satisfying the required staying period and room type, the room reservation request will be sent to the next process *Reserve*. The functional scenario of process *Reserve* indicates that a new reservation record will be successfully done and added into the data store *rlist*, if there exist enough rooms satisfying the received request.

Test cases are then generated from the functional scenario sequences of each path. Some sample test cases for covering the functional scenario sequence of *path 2* are listed in Figure 9.

Test case 1 stands for the execution of hotel reservation. The input value of variable *sel* is set to be 0. The staying period of room request is set to be *20110901-1*. The required amount of room is 1, respectively. The customer's name and email address are also generated.

By comparing the expected results derived from the involved functional scenarios and the execution results, we can determine the correctness of the *OHRS*. Once all the paths are covered by adequate test cases, a system module is thoroughly tested. Then we can proceed to test the high-level modules. Table 2 describes the test results of each path derived from the CDFDs of our *OHRS*.

In this table, number of functional scenario sequences of each path, inserted errors, detected errors and the corresponding error detection rate of each path derived from the CDFDs of the *OHRS* are listed. Specifically, for each path, we insert at least one error to its corresponding program statements. Since some paths derived from the CDFD share same program statements, corresponding errors are also shared by these paths. For example, 10 errors correspond to both *path 1* and *path 2* extracted from the CDFD. The detection rate of errors are described in the fifth column of Table 2.

The error detection rates of modules *HotelOpr*, *PaymentOpr* and *Portal* are *79.5%*, *91%* and *100%*, respectively. That is, 49 inserted errors are detected through the integration testing. The overall error detection rate is approximately *84.5%*.

Table 2. Test Result of the OHRS

path	functional scenario sequences	inserted errors	detected errors	detection rate
path_1	1	13	3	23.1%
path_2	2	30	23	76.7%
path_3	2	1	1	100%
path_4	1	2	2	100%
path_5	2	7	7	100%
path_6	2	4	4	100%
path_7	2	4	3	75%
path_8	2	5	4	80%

5 Related Work

Our approach inherits the model-based testing principle but differs in the aspects of specifications and the test data generation for integration testing. Classical formal specification languages, for instance, Z [3], the B [4] method and VDM [5], facilitate the precise requirements modeling, but lack effective mechanisms to describe the system architectures in an intuitive and rigorous manner, which makes integration testing and test result analysis difficult for practitioners.

Various integration testing approaches adopt the control flow diagrams, especially the UML sequence diagram for test case generation. In work [11], the authors transform the UML sequence diagram into the Sequence Dependency Diagram for test case generation. Similarly, the authors of work [7] and [12] combine the state-machine and UML sequence diagram for test data generation of integration testing. The work [6] also adopts the UML sequence diagram as the foundation for the integration testing of object-oriented programs.

The sequence diagrams facilitate the integration testing by clearly specifying the sequence of messages between different system components. However, since the sequence diagram only focuses on the control flows between system operations, other characteristics of the expected function, such as data flows, are represented by other diagrams or notations. As a result, practitioners need to deal with various diagrams or models, which degrades the efficiency and effectiveness of the integration testing. Distinguished from these approaches, our CDFD integrates the control flows and data flows in a unified diagram, which can be used as a firm foundation for rigorous test case generation and intuitive test results analysis for industry practitioners.

In the work [13] and [14], state charts are used as the foundation of test case generation for integration testing. As pointed out by the authors of [7], state charts are more powerful in unit testing rather than in integration testing. For large-scale software, state charts may be very large and complicated for analysis. To some extent, state charts are subject to the tester, which cannot precisely specify the expected functions, especially from the perspective of data structute definitions. Therefore, deriving rigorous test oracles from state charts for integration testing is also difficult.

Most researches on SOFL specification-based testing focus on unit testing level [15][10] while integration testing based on SOFL still demands more research efforts. In this paper, we further extends the CDFD-based integration testing approach proposed in [16] where a basic data flow path coverage criterion is put forward. Our new approach integrates the functional scenario-based test data generation with the basic paths coverage of CDFDs which allows for both rigorous test data generation and sufficient coverage on potential functional scenarios of the system under test.

6 Conclusion

In this paper, we propose a formal specification-based integration testing approach. Data flow paths are derived from rigorous CDFDs, and formal func-

tional scenario sequences are extracted from the paths as the foundation for test case generation. A functional scenario sequence coverage criterion and the corresponding algorithm are proposed and illustrated by a running example. The case study demonstrates the effectiveness and the feasibility of the approach.

To further promote this approach, we will continue exploring precise rules and effective techniques for constructing functional scenario sequences. Moreover, a powerful supporting tool is also one of future research projects.

Acknowledgment. This work is supported by SCAT foundation. It is also partly supported under the National Program on Key Basic Research Project (973 Program) Grant No. 2010CB328102; and NSFC No 61133001.

References

1. Bernot, G., Gaudel, M., Marre, B.: Software Testing based on Formal Specifications: A Theory and a Tool. Software Engineering Journal 6(6), 387–405 (1991)
2. El-Far, I.K., Whittaker, J.A.: Model-based Software Testing. Wiley (2001)
3. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall International (UK) Ltd. (1998)
4. Abrial, J.-R.: The B-Book. Cambridge University Press (1996)
5. Jones, C.: Systematic Software Development Using VDM, 2nd edn. Prentice Hall (1990)
6. Li, Z., Maibaum, T.: An Approach to Integration Testing of Object-Oriented Programs. In: Seventh Int'l Conf. on Quality Software, pp. 268–273 (October 2007)
7. Kansomkeat, S., Offutt, J., Abdurazik, A., Baldini, A.: A Comparative Evaluation of Tests Generated from Different UML Diagrams. In: Ninth ACIS Int'l Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, pp. 867–872 (August 2008)
8. Liu, S., Offutt, A.J., Ho-Stuart, C., Sun, Y., Ohba, M.: SOFL: A Formal Engineering Methodology for Industrial Applications. IEEE Transactions on Software Engineering (1), 24–45 (1998)
9. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer (2004)
10. Liu, S., Tamai, T., Nakajima, S.: A Framework for Integrating Formal Specification, Review, and Testing to Enhance Software Reliability. International Journal of Software Engineering and Knowledge Engineering 21(2), 259–288 (2011)
11. Samuel, P., Joseph, A.: Test Sequence Generation from UML Sequence Diagrams. In: Ninth ACIS Int'l Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, SNPD 2008, pp. 879–887 (August 2008)
12. Bandyopadhyay, A., Ghosh, S.: Test Input Generation Using UML Sequence and State Machines Models. In: Int'l Conf. on Software Testing Verification and Validation, ICST 2009, pp. 121–130 (April 2009)
13. Kansomkeat, S., Rivepiboon, W.: Automated-Generating Test Case using UML Statechart Diagrams. In: 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology, SAICSIT 2003, pp. 296–300. South African Institute for Computer Scientists and Information Technologists (2003)

14. Castro, L.M., Francisco, M.A., Gulías, V.M.: A Practical Methodology for Integration Testing. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2009. LNCS, vol. 5717, pp. 881–888. Springer, Heidelberg (2009)
15. Liu, S.: Utilizing Formalization to Test Programs without Available Source Code. In: The Eighth Int'l Conf. on Quality Software, QSIC 2008, pp. 216–221 (August 2008)
16. Chen, Y., Liu, S., Nagoya, F.: An Approach to Integration Testing Based on Data Flow Specifications. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 235–249. Springer, Heidelberg (2005)

Design and Implementation of a Tool for Specifying Specification in SOFL*

Mo Li[1] and Shaoying Liu[2]

[1] Graduate School of Computer and Information Sciences,
Hosei University, Tokyo, Japan
mo.li.3e@stu.hosei.ac.jp
[2] Department of Computer and Information Sciences,
Hosei University, Tokyo, Japan
sliu@hosei.ac.jp

Abstract. Structure Object-oriented Formal Language (SOFL) is not just a formal language for writing formal specification. It is also an approach and a methodology. SOFL provides a three-step approach for modelling a software system using formal specification. Writing specification can be realized as the most important and fundamental task in this modelling approach. In practice, the activity of writing specification is error-prone, especially the activity of specifying formal specification. We think there are two reasons that cause the difficulty of specifying specification. One reason is that some specifiers may not be familiar with the formal notations used in SOFL, especially the mathematical notations. And the other reason is that there is no tool to guide the specifiers to write specification and make the specifying process easy. In this paper, we show a prototype of a tool that can provide the specifiers with a strong support in the process of specifying specification. This tool provides an integration environment for specifying all kinds of specifications used in SOFL approach, including informal specification, semiformal specification, formal specification, CDFD, and class. And the tool also provides the function to organize the specifications of a same software system.

Keywords: formal method, specification, modelling approach, tool.

1 Introduction

Formal methods have been recognized as an effective approach for software development. One of the products of formal method is formal specification. Formal specification can describe a software system precisely. The requirement of the software system is specified by formal notations, usually mathematical notations, in the formal specification. The content and structure of formal specifications is different over different formal specification languages. Several formal specification languages exist, like VDM-SL [4], Z [5], Object-Z [6], and so on. SOFL

* This research is supported in part by NII Collaborative Program, SCAT Research Foundation, and Hosei University. It is also partly supported by China 973 program under Grant No. 2010CB328102 and NSFC under Grant Nos. 61133001, 60910004..

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 44–55, 2013.

(Structure Object-oriented Formal Language) [1] is one of the these formal languages.

SOFL is not only a formal language for writing formal specification, but also an approach and a methodology. SOFL provides a three-step approach for modelling a software system using formal specification. And a lot of techniques have been created for verifying and validating the formal specification and corresponding implementation program based on formal specification. The combination of SOFL three-step modelling approach and relative verification and validation techniques provide a framework of the entire software system development process. The soul of this framework is the formal specification. It is the final product of modelling process and the basis of following verification, validation and implementation. Writing formal specification can be realized as the most important and fundamental task in SOFL approach.

In practice, specifying formal specification is an error-prone activity. We think there are two reasons that cause the difficulty of specifying formal specification. One reason is that some specifiers may not be familiar with the formal notations used in SOFL, especially the mathematical notations. And the other reason is that there is no tool to guide the specifiers to write specification and make the specifying process easy. The tool support is actually very important. Since there is a lot of special concepts in SOFL, tool support can facilitate the specifiers to deal with these concept. Typically, CDFD (Conditional Data Flow Diagram) is an unique concept in SOFL. Drawing CDFD is required when specifying formal specification. A specific tool that can be used to draw CDFD directly will be very helpful.

In this paper, we show a prototype of a tool that can provide the specifiers with a strong support in the process of specifying specification. This tool provides an integration environment for specifying all kinds of specifications used in SOFL approach, including informal specification, semiformal specification, formal specification, and class. And the tool also provides the function to organize the specifications of a same software system. The prototype is implemented in C# programming language under the environment of Microsoft Visual Studio 2008.

The rest of this paper is organized as follows. We introduce some special concepts of SOFL in Section 2. Specifically, we also explain the three-step modelling approach in this section. In Section 3, we describe the major functions of the tool and explain how these functions support the modelling approach. We demonstrate the architecture of the tool in Section 4 and introduce the implementation in Section 5. Section 6 is related work. And finally, we conclude in Section 7.

2 SOFL Three-Step Modelling Approach

In the first step of three-step modelling approach, the informal specification should be specified. The informal specification is written in natural language and is the simplest specification in SOFL. It is used to communicate with the end users or domain experts. The basic unit of informal specification is "*module*".

A module is a group of descriptions of functions. It is like a component in software system.

The second step of the modelling approach is to build semiformal specification. The semiformal specification consists of two aspects. One is process specification, and the other is CDFD. The process specification is plain text specification.The basic unit in process specifications is "process". A process is an independent operation that processes data, and different processes contact with each other via data flows. A group of processes and their relationship are integrated to compose a "module". A module can be considered as a higher level process. Each process can also be decomposed into a lower level module. We use "module" and "specification" changeably in the following paper. Usually these two terms indicate the same thing, namely the process specification of a module.

The counterpart of process specification is CDFD, a graphic specification. For each module, there is a corresponding CDFD. The CDFD uses visual notation to express the relation between different processes that are included in formal specification. A process in a CDFD is treated as a transition and a data flow as a token. When all the input data flows of the process become available, the process will be enabled and executed. The CDFD is both a formal and intuitive notation that is suitable for describing the process specification. Note that, even the semiformal specification is composed by process specification and corresponding CDFD. Drawing a CDFD for a semiformal module is not required by SOFL.

The third, or the final step of the three-step approach is to specify formal specification. The formal specification has the same structure of semiformal specification. It is also composed by CDFD and corresponding process specification. The difference is that the notations used in formal process specification are formal notations, and drawing CDFD for a formal module is not an optional. SOFL requires the specifiers to draw CDFD for each formal module.

Except for the informal, semiformal and formal specification, there is another kind of specification include in the SOFL specifications. It is the "*class*" specification. Since SOFL is an object-oriented formal language, the most important concept of object-oriented design, *class*, is included in the SOFL specification.

The finished specifications should be verified and validated. The incorrect parts of the specifications will be corrected or modified. And then the modified specifications will be verified and validated again. This process will repeat. It is similar to the cycle of software development.

3 Design of the Tool

Based on the previous introduction, the entire process of modelling software system by using SOFL can be divided into two stages. The first stage is to specifying specifications used in SOFL, and the second stage is to verify and validate the formal specifications finished in the first stage. As shown in Figure 1, our tool is also separated into two parts and each part corresponds to one stage mentioned above. In this paper, we focus on describing the first part of the tool, namely the part that supports specifying specifications. The tool provides

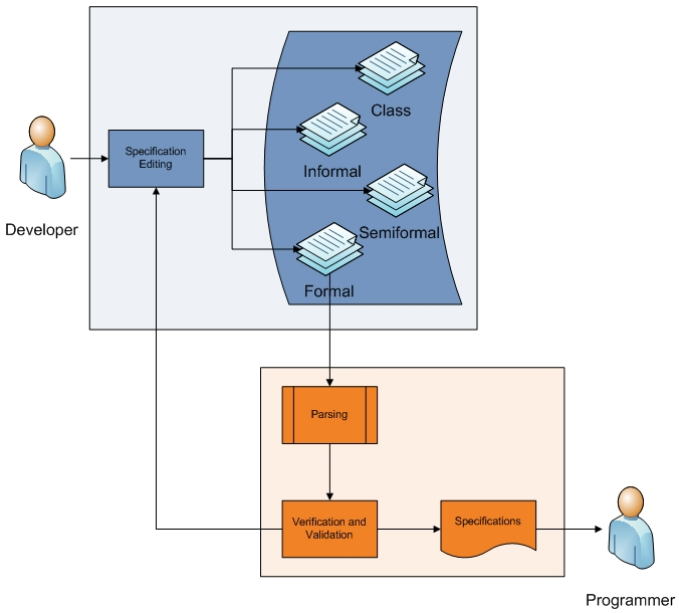

Fig. 1. The components of the tool

not only a plain text editor, but also a group of functions that facilitate the users in the specifying process expediently.

According to the three-step modelling approach, for each target system, specifiers should construct the informal specification first, then the semi-formal specification, and finally the formal specification. In order to support this specifying process, our tool provides following 10 major functions:

1. creating a software project
2. adding an informal module
3. specifying informal specification
4. adding a semiformal module
5. specifying semiformal process specification
6. adding a formal module
7. specifying formal process specification
8. drawing CDFD
9. specifying class
10. export specifications, including CDFDs

Most of functions listed above correspond to specifying specifications in SOFL: informal specification, semiformal specification, formal specification, and class. Specially, semiformal and formal specification include process specification and CDFD. The difference is that the CDFD in semiformal specification is optional, but the CDFD in formal specification is required. Since the CDFD is a different presentation of corresponding formal process specification, keeping consistency between CDFD and process specification is one of our major concern.

4 Architecture of the Tool

The specifications are the final products of using our tool. To help users organizing the working space, all of the specifications that describe the same software system is grouped. The combination of the specifications is called a "*project*". Figure 2 shows the hierarchy of the specifications. The CDFD with shadow box indicates that the CDFD in semiformal specification is optional.

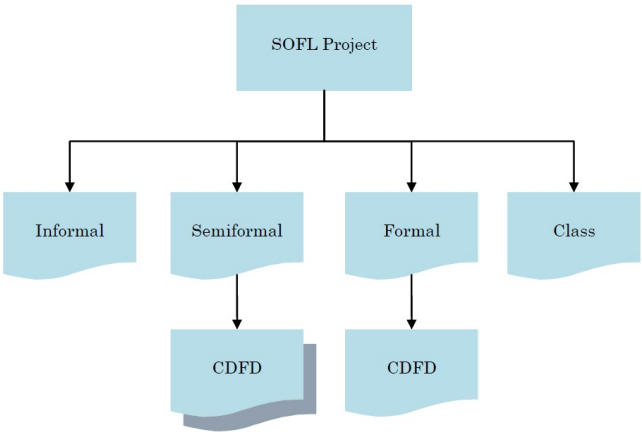

Fig. 2. The hierarchy of SOFL specifications

All of the specifications including CDFD are saved as XML files in our tool. We use the module as the basic unit to create a XML file. For example, if users add a new formal module to the SOFL project, two new independent XML files will be created to save the CDFD and process specification, respectively. All the specifications saved in XML files can be exported as other files format for later reference. The XML files will also be used as bases for verification and validation.

Table 1. The files used to save specifications

No.	Suffix	Description
1	.soflproject	save the hierarchy of the SOFL project
2	.ifModule	save the specification of a informal module
3	.sfModule	save the process specification of a semiformal module
4	.sfCDFD	save the CDFD of a semiformal module, the content can be empty
5	.fModule	save the process specification of formal module
6	.cdfd	save the CDFD of a formal module
7	.classSpec	save the class definition

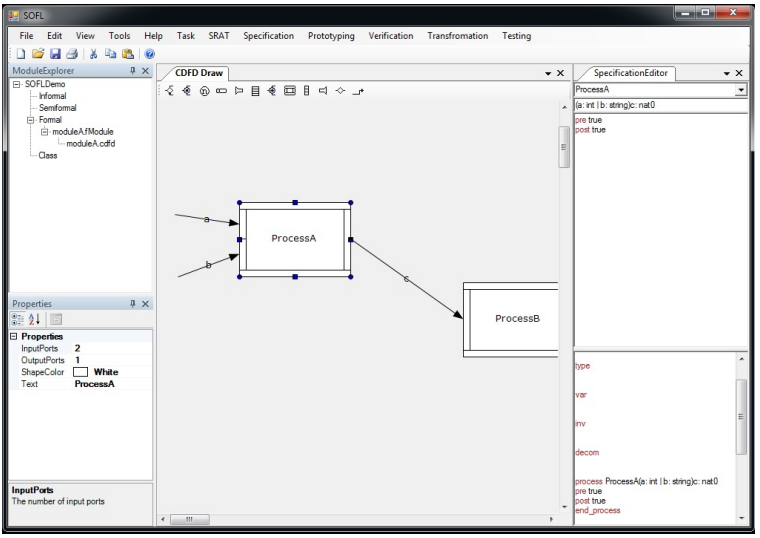

Fig. 3. The *Viewer* for specifying formal specification

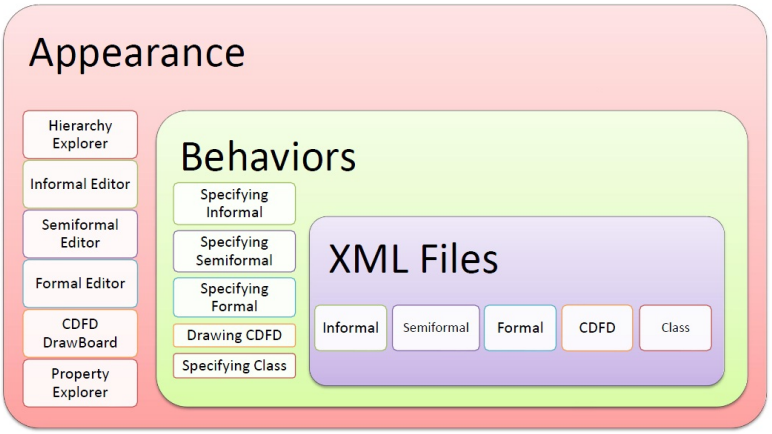

Fig. 4. The architecture of the tool

Expect for the XML files, which are used to save the specifications, one specific XML file is used to store the hierarchy of all the specifications, namely, the hierarchy of SOFL project. The internal structure of this XML file is consistent with the structure shown in Figure 2. Table 1 lists all the XML files that will be created by the tool. "Suffix" column shows the suffix of the XML file, and "Description" column explains the content of the XML file.

In our tool, we use different combinations of small windows to provide users with the interfaces for different tasks. The small windows in the tool are called "*Explorers*". Each explorer focuses on presenting one aspect of a specific task. An combination of explorers is called an "*Viewer*" in the tool. For example, Figure 3 shows the default viewer for specifying formal specification. Except for the default viewers, users can rearrange the layout of the explorers to build customized viewers for different tasks. The users who used to use Eclipse or Visual Studio will be familiar with such kind of interface.

Figure 4 demonstrates the architecture of the tool. Users use the functions provided by the tool through different viewers. All the specifications and CDFDs specified by the users are stored in a group of XML files.

5 Formal Specification Editor

There is not doubt that the formal specification is the most important part in SOFL. Almost all of the verification and validation techniques are designed based on formal specification. But specifying formal specification is the most challenge task when using SOFL. Even the skilful developer would make mistake when specifying formal specification. And the formal notations in formal specification will sometime confuse the readers, too. In order to help the users to specify and understand the formal specification, SOFL uses a graphic specification called CDFD to simplify the process specification. The CDFD can be realized as a overview of the corresponding process specification. When specifying a formal specification, the user can draw CDFD first, and use the CDFD as a guideline to write process specification.

In our tool, two XML files will be created when the user adds a new formal module to a software project. Figure 3 shows the default viewer for specifying formal specification. This viewer includes four explorers. The two explorers at the left hand side are "*Hierarchy Explorer*" and "*Property Explorer*". The center explorer is "*CDFD Drawboard*", and the explorer at the right hand side is "*Formal Editor*".

The "*Hierarchy Explorer*" displays the hierarchy of the project. The root node of the tree structure in the "*Hierarchy Explorer*" presents the project's name. Under the project's name there are four second level nodes, and each node corresponds to a specific specification type. Different kinds of specifications, namely modules are listed under corresponding node. The viewer for specifying formal specification will open when node presenting a formal module being double clicked.

5.1 Drawing CDFD

Drawing CDFD is the first step to specify a formal module. Of course drawing CDFD first is not required, but we strongly recommend it. In the tool, the "*Property Explorer*" and "*CDFD Drawboard*" work together to help users draw CDFD. On the top of the explorer "*CDFD Drawboard*", there is a tool bar.

Each button in this tool bar corresponds to a component in CDFD. Users can add a figure of a component onto the board by clicking the corresponding button. The figure of a component is called an "object" on the board. User can move the objects, resize the objects like using other popular drawing tool. Two objects can be connected via data flow. The point at which object and data flow are connected are called "connector". In different figures, the numbers of connectors are different. For example, considering the two processes "ProcessA" and "ProcessB" shown in Figure 3, there are two connectors at the left side of "ProcessA" but only one connector at the left side of "ProcessB". This is because "ProcessA" has two input ports, while "ProcessB" has only one input port.

Some components of CDFD have their own name or specific properties. In order to edit the name or properties of an object in the board, users can just simply select the object by clicking it, and the corresponding properties will be list in the "*Property Explorer*" automatically. Users can change the name or values of properties, and the modification will change the figure of the object directly. For the sake of space, we just demonstrate one example here. For instance, the selected object in Figure 3 is "ProcessA". It has a name and properties such as input port number and so on. All of the properties are listed in the "*Property Explorer*" at the left-bottom corner. We can see that the input port number is 2, output port number is 1, and name is "ProcessA". In addition, we add an additional property for this component, "ShapeColor". Users can select a predefined color to highlight the object.

5.2 Specifying Formal Process Specification

The "*Formal Editor*" explorer in Figure 3 is used to specify the formal process specification, the counterpart of CDFD. The entire explorer is divided into two sections. One is for editing and the other is for displaying. The editing section is separated into three parts. For top to bottom, the three parts are "Component List", "Head Displayer", and "Content Editor". The "Component List" is a drop-down list and all the components of a process specification are listed in it. The component includes *Constant Declaration*, *Type Declaration*, etc. Users can select one specific component to edit each time. Note that the processes defined in the process specification are also listed in this drop-down list. Users can also select a process to edit. Once users select a specific component in the "Component List", the "Head Displayer" will display the head declaration of this component. And the users can edit the content of the selected component. Everything that users type in the "Content Editor" will be displayed in the displaying section automatically. For example, the component selected to edit in Figure 3 is process "ProcessA". We can see the head declaration of "ProcessA" is displayed in the "Head Displayer", and the content in "Content Editor" is also presented in displaying section.

Table 2. The events that may effect the consistency

No. of Event	Event	Effected Process
1	change process's name	1
2	change process's input port number	1
3	change process's output port number	1
4	change data flow's name	2
5	change data flow's type	2
6	add a new process to CDFD	1
7	delete a process from CDFD	1
8	add a new data flow to CDFD	usually 0
9	delete a data flow from CDFD	2
10	connect a process and a data flow	1
11	disconnect a process and a data flow	1

5.3 Keeping Consistency Mechanism

One of the mistakes made in specifying formal specification is inconsistency between CDFD and corresponding process specification. In our tool, we provide a well designed mechanism to keep the consistency between CDFD and process specification. We defined total 11 events that can effect the consistency. These 11 events are listed in Table 2. The column "Event" is the description of the event, and the column "Effected Process" is the number of processes whose definition will be changed by the event. For instance, the forth raw of the table indicates that the event of changing the name of a data flow will effect two processes in the process specification. The two processes are connected by the data flow. When this event happen, the corresponding definition of these two processes in the process specification should be changed.

In order to make this mechanism work well, we require all of these events must happen in the process of drawing CDFD. It means all the change and modification described in Table 2 must be done through "CDFD Drawingboard" and "Property Explorer". And the content of process specification will be modified automatically. Users cannot do these change or modification in the "Formal Editor". Of course we can provide users with a check list based on Table 2 for inspecting the consistency, but we think build this mechanism into the tool will give the users a very good guide to specify the specification and it can avoid mistakes in inspecting the consistency.

5.4 Specifying Semiformal Specification

Specifying semiformal specification is similar to specifying formal specification. Two specifications have almost same structure. In our tool, the default viewer for

specifying semiformal specification is similar to the default viewer for specifying formal specification. Figure 5 is the snapshot of default viewer for specifying semiformal specification. Compare to Figure 3, the "CDFD Drawboard" and "Property Explorer" disappear. This is because drawing CDFD is optional in the process of specifying semiformal specification. And the "Formal Editor" is replaced by "Semiformal Editor". The "Semiformal Editor" looks like "Formal Editor". The only difference is the tool bar in "Semiformal Editor". We can see from Figure 5 that there are two buttons in the tool bar. The two buttons correspond to adding a *process* and a *function* to the "Component List" respectively. When specifying formal specification, adding a process to specification is an event that will effect the consistency between CDFD and process specification. Therefore, a process can be added to the process specification by adding a process object to the CDFD. But in the process of specifying semiformal specification, we do not force the users to draw CDFD, drawing CDFD is just an option. If the users do not want to draw CDFD, they can use the two buttons in the "Semiformal Editor" to add process or function definition to the semiformal process specification. In this case, there is no need to check the consistency between semiformal process specification and its corresponding CDFD, and we do not provide the relative functions in the tool.

If users want to draw CDFD for a semiformal process specification, he or she can just simply double click the node with the suffix ".sfCDFD", and the same "CDFD Drawboard" and "Property Explorer" shown in Figure 3 will be opened. Note that the occurrences of the events listed in Table 2 will not be presented in "Semiformal Editor".

6 Related Work

The tool described in this paper provides several functions that can make the process of specifying SOFL specification easier. There is another tool that can support specifying SOFL formal specification is introduced in [7]. Compare to our tool, this tool does not provide the function to check and keep the consistency between the CDFD and formal process specification. Furthermore, the flexibility of this tool is limited, while our tool is designed not only for providing support for specifying specification, but also for utilizing the latest published verification and validation techniques. The prototype introduced in [8] is designed to exact functional scenarios from SOFL formal specification. It reads formal specification from XML files and generate all possible functional scenarios. The functional scenario is the basic in several verification and validation techniques.

Several tools have been to support different formal languages or formal methods. Overture [9] is a community-based project of open-source tools to support modelling and analysis in the design of software systems using VDM. It provides several functions such as editing, checking, debugging, etc. B4Free [10] is a set of tools for the development of B formal models. B4Free offers a graphic representation and numerous functionalities to present and manage projects in B language. And it also provides automatic management of dependencies between

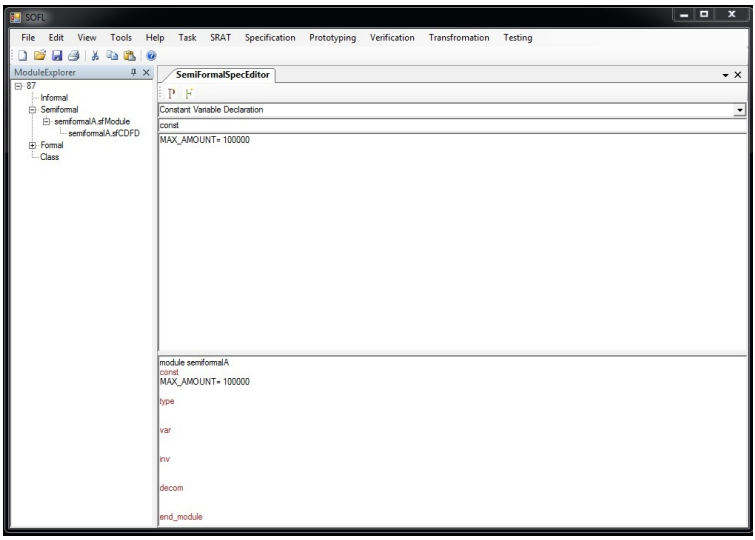

Fig. 5. The default *Viewer* for specifying semiformal specification

B components. The Rodin Platform [11] is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof. PiZA [12] is an animator for Z. It translates the Z specifications into Prolog to generate output variables.

7 Conclusions and Future Work

In this paper, we present the prototype of a tool that supports the entire specification specifying process when using SOFL three-step modelling approach. The tool offers customized editor for each kind of specification and numerous functions facilitating specifiers. These functions include manage SOFL project, checking consistency between CDFD and formal process specification, etc. All of the specifications specified by users and the hierarchy of project are saved in XML files. The XML files of formal specifications will be used as basis for later verification and validation. If the users want to distribute the specifications, the specifications can be exported as different formats. The process specification can be experted as MS Word file or PDF file, and the CDFD can be exported as JPEG file or BMP file.

As show in Figure 1, the goal of our tool is not to provide an editing environment only. We want to provide an entire environment for the SOFL three-step modelling approach and the following verifying and validating process. Obviously, building a powerful specifying tool is a good start. On the one hand, verification and validation cannot be performed without formal specification. On the other hand, a well defined and uniform formal specification file can simplify the process of tool support verification and validation. In the future, our

work can be separated into two parts. The first part is to refine the exiting tool. Enhance the usability of the tool. The second part of our work is to build a parser for formal specification. Almost all of the verification and validation techniques are based on analysing the formal specification. Therefore, build a parser is necessary.

References

1. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer (2004) ISBN 3-540-20602-7
2. Liu, S., Sun, Y.: Structured Methodology + Object-Oriented Methodology + Formal Methods: Methodology of SOFL. In: 1st IEEE International Conference on Engineering of Complex Computer Systems, pp. 137–144. IEEE Press, Ft. Landerdale (1995)
3. Liu, S., Shibata, M., Sato, R.: Applying SOFL to Develop a University Information System. In: 6th Asia-Pacific Software Engineering Conference, pp. 404–411. IEEE Press, Takamatsu (1999)
4. Dawes, J.: The VDM-SL Reference Guide. Pitman (1991)
5. Diller, A.: Z: An Introduction to Formal Methods. John Wiley & Sons (1994)
6. Meira, S.R.L., Cavalcanti, A.L.C.: Modular Object-Oriented Z Specifications. In: 5th Annual Z User Meeting on Z User Workshop, pp. 173–192. Springer, London (1991)
7. Liu, S.: Integrating top-down and scenario-based methods for constructing software specifications. In: 8th International Conference on Quality Software, pp. 105–113. IEEE Press, Oxford (2008)
8. Li, M., Liu, S.: Automatically Generating Functional Scenarios from SOFL CDFD for Specification Inspection. In: 10th IASTED International Conference on Software Engineering, Innsbruck, Austria, pp. 18–25 (2011)
9. Overture: Formal modelling in VDM, `http://www.overturetool.org/`
10. B4Free, `http://www.b4free.com/index-en.php`
11. Event-B.org, `http://www.event-b.org/`
12. Hewitt, M.A., O'Halloran, C.M., Sennett, C.T.: Experiences with PiZA, an animator for Z. In: Till, D., Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 37–51. Springer, Heidelberg (1997)

Development of a Supporting Tool for Formalizing Software Requirements

Xi Wang and Shaoying Liu

Department of Computer Science, Hosei University, Japan

Abstract. Formal specification precisely documents expected behaviors of the system under construction, which provides a firm foundation for ensuring software quality and facilitating software maintenance. However, describing software requirements in formal specifications remains a challenge for practitioners and becomes one of the obstacles toward widespread use of formal methods in industry. To deal with the problem, this paper describes an interactive tool that assists designers in software requirement formalization. It retrieves informal requirements from the designer and automatically generates the corresponding formalization result. Based on a knowledge base stored in XML files, the tool implements a core engine for producing comprehensible guidance. By conducting a case study on a banking system, the effectiveness of the tool is shown.

1 Introduction

Statistic data shows that formal methods largely improve the quality of software products by conducting verification based on formal specifications [1][2][3]. However, according to our experience with industry, the development community refuses to adopt such technique in real practice, which announces a considerable gap between formal methods and software development process. One of the major reasons is the difficulty in formalizing software requirements into formal specifications, especially in describing functions in formal expressions (such as pre- and post-conditions of operations), which requires sophisticated math background, as well as tremendous patience and care. As complexity rises, organizing large amount of informal ideas in a designated formal manner without missing any information becomes even harder. Thus, having realized the fact that building formal specifications is a time-consuming and error-prone activity, and needs a long-term training, practitioners still prefer conventional approach to developing software which seems more practicable and controllable.

Describing functions in formal expressions comprises two tasks: function design, which mainly depends on human intelligence, and formalization of the function description, which focuses on tedious syntax problems and would cause more and more errors as the complexity of the function rises. Therefore, we believe that the formal specification technique will be acceptable to practitioners if the first task can be guided and the automation level of the second task can be enhanced. To this end, this paper describes a tool that aims at generating formal expressions automatically with obtained informal ideas from developers

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 56–70, 2013.

for formal specification construction. It is composed of a knowledge base and a core engine. By conducting a proposed knowledge retrieval algorithm, the core engine retrieves appropriate knowledge from the knowledge base to retrieve desired functions from the designer through interactions in natural language and formalizes such information.

The knowledge base is built on the basis of the pattern system proposed in our previous work [4]. In the pattern system, a set of patterns are pre-defined with a same structure, each providing a solution to the formalization of a kind of function. These patterns are organized in a hierarchical manner in the pattern system and the application of an individual pattern may need the application of others. In order to simplify the implementation of the core engine and avoid frequent modification of it, the application process of the pattern system, rather than the pattern system itself, is treated as knowledge and stored in the knowledge base.

Finite State Machine (FSM) is employed to represent the knowledge in the knowledge base since applying the pattern system is to interact with the developer for the target formal expression and FSM is a formal modeling language easy to be manipulated in an automated way for describing interactive behaviors. Furthermore, as a classic language with a long development history, it includes many mature techniques that can be directly adopted. We depict the application process in one FSM model and introduce some symbols into the model to enable the representing of all its aspects.

When implementing the tool, XML files are used to store the knowledge base where a set of tags are created to identify different kinds of information in the FSM model and the informal explanation of some symbols are attached for generating comprehensible guidance. Based on the XML file, the core engine extracts necessary information by recognizing the pre-defined tags before conducting the proposed knowledge retrieval algorithm to retrieve and formalize informal ideas.

To evaluate the effectiveness of the tool, we conduct a case study on modeling the behavior of an example banking system. The result shows that developers who are not familiar with formal notations can also write formal expressions using the tool since they will only be required to make decisions on function design issues and all the annoying syntactic work is handled by machines. As soon as the tool retrieved necessary details of the desired functions, a corresponding formal expression will be generated.

It should be noted that the underlying theory of the tool is language independent, we use SOFL [5][6][7] as an example formal notation to demonstrate how the tool works.

The remainder of this article is organized as follows. Section 2 overviews the related work. Section 3 presents the design and the implementation of the tool. A case study on an example banking system is given in Section 4. Finally, Section 5 concludes the paper and points out the future work.

2 Related Work

Many researches have been done on formal specification construction method. S. Liu [8] proposes an approach to constructing software specifications by

integrating top-down and scenario-based methods. J. Ding et al. [9] propose a refinement method based on a set of formal refinement patterns for software architecture design using Software Architecture Model (SAM). Stephney et al. [10] describe a pattern language for using notation Z in computer system engineering. S. Vadera et al. [11] describe an interactive approach for producing formal specifications from English specifications. Konrad et al. [12] create real-time specification patterns in terms of three commonly used real-time temporal logics based on an analysis of timing-based requirements of several industrial embedded system applications and offer a structured English grammar to facilitate the understanding of the meaning of a specification.

A majority of these methods cope with formal specification construction at a more abstract level compared with formal expression generation. Others that are intended to support the construction process at the bottom level can be divided into two kinds. One is automatic or semi-automatic transformation from informal descriptions into formal specifications using NLP (Natural Language Processing). However, since NLP is still considered as a unreliable technique, the correctness of the transformation result is hard to guarantee. The other is the introduction of specification patterns to provide solutions to re-occurred problems in writing formal expressions. But they are only capable of facilitating the formalization of some typical properties and before the suggested formal expression is generated, developers have to read through and understand all of the patterns to select an appropriate one and apply it to the specific problem. Thus, despite of the rising interest, this area is still lack of tool-support.

By contrast, informal descriptions is not treated as the input of our tool. The desired functions are obtained by gradually clarifying informal ideas with human involvement. Thus, the performance of the tool will not be affected by NLP technology. Besides, the knowledge stored in the tool is invisible to developers and designed for machines to generate guidance, which provides a possibility to automate the generation process of formal expressions.

3 Tool Design and Implementation

Before presenting the design and implementation of the tool, SOFL is briefly introduced which serves as the example formal notation in this paper. Readers who wish to understand the details can refer to the SOFL book [5].

3.1 SOFL

SOFL (Structured Object-Oriented Formal Language) is a formal engineering method providing both a formal language and a practical method for developing software systems.

SOFL specification language integrates Data Flow Diagrams, Petri nets and VDM-SL. A specification in SOFL is composed of a set of modules in a hierarchical manner reflecting the architecture of the real system under construction. Each module is associated with a CDFD (Condition Data Flow Diagram) representing the behavior of the module and encapsulates necessary data, including

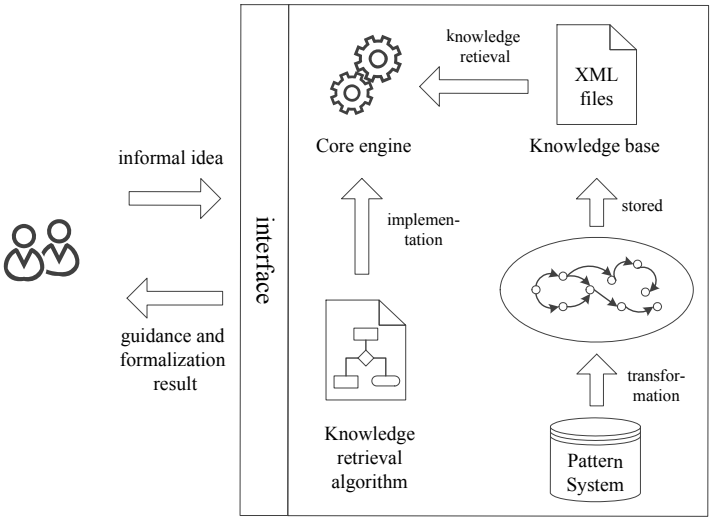

Fig. 1. The overall structure of the tool

types and variables, and processes used in the CDFD. These processes are connected by their interfaces and each of them describes the relation between its associated input and output in terms of pre- and post-conditions.

Since this paper concentrates on formal expression generation process, we assume that the hierarchical relation between specification modules is determined, as well as their attached types, variables and process interfaces, and the tool is aimed at assisting developers in writing pre- and post-conditions.

3.2 Tool Design

The description of the tool starts from an overview on the design of the tool and the included components will then be presented in details respectively.

Overview. Figure 1 shows the overall structure of the tool.

It is composed of two major components: a knowledge base and a core engine. For each unit function of a pre- or post-condition, by retrieving appropriate knowledge from the knowledge base, the core engine produces guidance to capture informal ideas from the developer and generates corresponding formalization result. The knowledge in the knowledge base is represented in a FSM model transformed from the pattern system and a set of tags are created to store the FSM model in XML files. By recognizing these tags, the core engine extracts necessary information from the XML files and implements a proposed knowledge retrieval algorithm to interact with the designer through the interface of the tool.

Knowledge Base. As the critical component of the tool, the knowledge base is built on the basis of a pattern system where each pattern is defined as follows to be applied by machines to guide the formal description of a kind of functions.

name the identity of the pattern
explanation summary of the functions that the pattern is able to describe
constituents a set of elements that must be specified to apply the pattern and a set of rules for guiding the specifying process
solution a set of rules for generating formal expressions according to the specified elements

Figure 2 shows an example pattern where $dataType(x)$ denotes the data type of variable x and $constraint(obj)$denotes certain constrains on object obj.

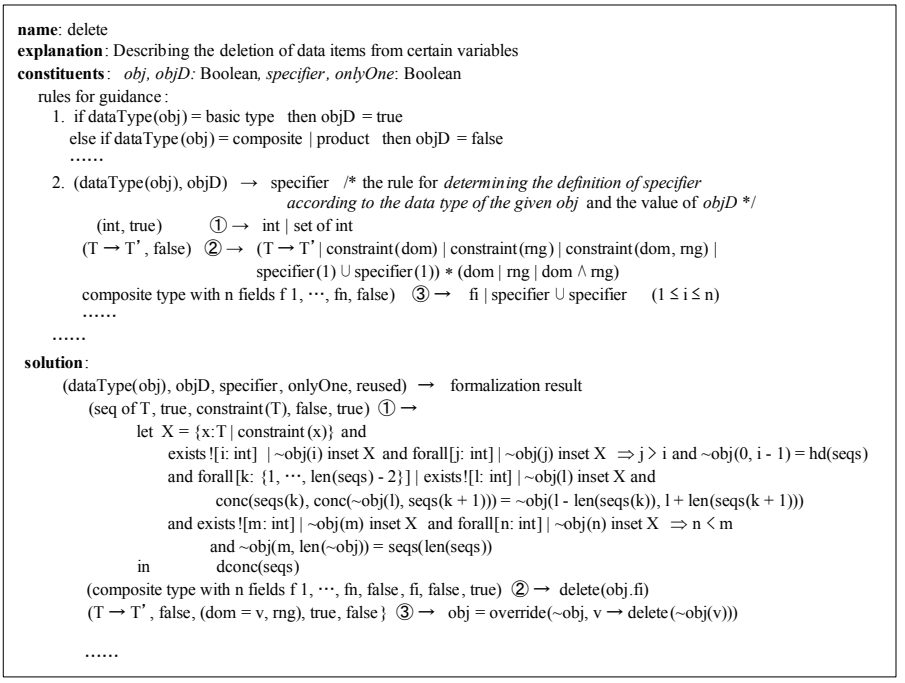

Fig. 2. Pattern *delete*

Item *explanation* and *constituents* reveal that the pattern *delete* is applied for describing deletion functions which need four elements to be specified: *obj* denoting the object from which the required data items are deleted; $objD$ denoting deletion from parts of *obj* if evaluated as false and denoting deletion directly from *obj* if evaluated as true; $specifier$ denoting the constraints on the data items to be deleted; *onlyOne* denoting more than one data item need to be deleted if evaluated as false.

Besides for listing necessary elements in *constituents*, rules for guiding the specifying process of these elements are also provided in "rules for guidance". For example, if the given *obj* satisfies certain *if* condition in rule 1, element $objD$ will be designated with a value accordingly. Otherwise, the developer will be asked to make a decision on the value of $objD$ based on the understanding of the intended function. In rule 2, the definition of $specifier$ is determined by the given *obj* and $objD$. For instance, mapping 1 indicates that if the given *obj* is an integer and $objD$ is evaluated as true, element $specifier$ can be either an integer or a set of integer.

Item *solution* is defined as a mapping where the formalization result can be generated depending on five factors: the data type of the given *obj*, the values designated to $objD$, $specifier$ and *onlyOne*, and whether the pattern is reused. It can be seen from mapping 2 and 3, there exist informal expressions in the formalization result comprised of pattern names and parameters. For instance, expression "$delete(obj.fi)$" in mapping 3 is a combination of pattern name "*delete*" and a parameter "$obj.fi$", which represents the formalization result generated by applying pattern *delete* with its first element *obj* designated as $obj.fi$. Such an application process is conducted within another upper-level one and its involved pattern is identified as being reused.

The well-defined individual patterns are organized hierarchically by classifying functions, which forms a pattern system with categories and patterns in different levels. After analyzing a large number of typical formal specifications from industry, we found that they describe system behaviors mainly using three basic functions and the top level of the pattern system is accordingly designed with three categories: *Relation* patterns for describing relations between objects, *Recreation* patterns for describing modifications on certain objects and *Retrieval* patterns for describing certain system variables in formal expressions. These categories are further classified into sub-categories or patterns. For example, *Modification* and *Rearrangement* are two of the sub-categories within *Recreation* category. The former includes pattern *add*, *alter* and *delete* while the latter includes pattern *sort*, *group* and so on.

To expand the range of expressive functions, the pattern system needs to be updated by introducing new patterns and adding rules in *solution* items of existing patterns. Therefore, if it is directly stored in the knowledge base, the knowledge retrieval algorithm would be increasingly complex and frequently modified. To this end, the application process of the pattern system is treated as knowledge in our tool which describes how the target formal expression is generated through an interaction process, since the semantics of interaction processes will remain the same when they are updated.

Before describing the details of the knowledge, its representation language is first introduced. Since the knowledge is designed for machines to use and remains invisible to designers, whether machine processing can be effectively supported is the major concern of the representation language. As a mature technique for modeling interactive system behaviors, FSM is easy to be manipulated in an

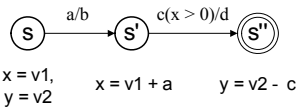

Fig. 3. An example FSM

automatic manner and used to represent the knowledge of the tool. We define it as follows.

Definition 1. *A FSM (Finite State Machine) is a 9-tuple* $(Q, q0, F, VP, I, G, \varphi, \delta, \lambda)$ *where* Q *is a non-empty finite set of states,* $q0 \in Q$ *is the initial state,* $F \subset Q$ *is the set of accept states,* VP *is a set of triples* (V, V', θ) *where* V *is the finite set of system variables,* V' *is a set of values and* $\theta : V \longrightarrow V'$ *indicates the value of each* $v \in V$*,* I *is the finite set of symbols,* G *is the finite set of guard conditions,* $\varphi : Q \longrightarrow VP$ *is the state function indicating the values of the involved variables on each state,* $\delta : Q \times (I \times \mathcal{P}(G)) \longrightarrow Q$ *is the transition function relating two states by input and guard conditions,* $\lambda : Q \times (I \times \mathcal{P}(G)) \longrightarrow I$ *is the output function determining output based on the current state and input.*

In a FSM, each state denotes certain stage of the guidance production process, each $i \in I$ denotes a symbol for composing inputs and outputs, and each $g \in G$ denotes a constraint. Figure 3 shows an example FSM EA where

$$
\begin{aligned}
&Q_{EA} = \{s, s', s''\}, q0_{EA} = s, F_{EA} = \{s''\},\\
&VP_{EA} = \{vp_1, vp_2, vp_3\} \text{ where } vp_1 = (\{x, y\}, \{v1, v2\}, \{x \to v1, y \to v2\}),\\
&\ vp_2 = (\{x, y\}, \{v1 + a, v2\}, \{x \to v1 + a, y \to v2\}),\\
&vp_3 = (\{x, y\}, \{v1 + a, v2 - c\}, \{x \to v1 + a, y \to v2 - c\}),\\
&\ I_{EA} = \{a, b, c, d\}, G_{EA} = \{x > 0\}, \varphi_{EA} = \{s \to vp_1, s' \to vp_2, s \to vp_3\},\\
&\ \delta_{EA} = \{(s, (a, \varnothing)) \to s', (s', (c, \{x > 0\})) \to s''\},\\
&\lambda_{EA} = \{(s, (a, \varnothing)) \to b, (s', (c, \{x > 0\})) \to d\}
\end{aligned}
$$

It demonstrates values of system variables on different states by the attached equations. When EA is activated and stays on state s, system variable x and y are initialized as $v1$ and $v2$ respectively. After receiving an input a, symbol b is set as output and EA is transferred to state s' where the value of variable x is modified to "$v1 + a$". Finally, state s'' will be reached when receiving input c if property "$x > 0$" establishes. Meanwhile, variable y will be set as "$v2 - c$".

Definition 2. *Given a FSM* A *and a state* $s \in Q_A$*,* $Acc_A(s) \in \mathcal{P}(I \times \mathcal{P}(G))$ *is an acceptable set on* s *iff* $\forall_{acc \in Acc_A(s)} \cdot (\exists_{s' \in Q_A} \cdot \delta(s, acc) = s')$.

For each state s in A, $(i, G) \in Acc_A(s)$ means that there exists a transition originated from s, which will be activated if input i is received and each $g \in G$ can be satisfied. Note that $(i, \varnothing) \in Acc_A(s)$ and $(\varepsilon, G) \in Acc_A(s)$ may also establish. The former indicates a transition that will be triggered on s when

Table 1. Symbols involved in FSM models

symbol	definition
$\sum S$	Providing items in set S for the designer to choose from
$\&item_i$	Selection of item $item_i$
$\#k$	Asking for pressing key k
$req(x)$	Element or variable x is required to be designated with a value
$legal(i)$	Input symbol i is written in defined variables and formal notations
$patterns$	The variable indicating all the patterns in use
$pattern$	The variable indicating the pattern currently being applied
$elems$	A variable of sequence type that holds the values of the elements of $pattern$
$\#mM.v$	The value of variable v in module M
$formalExp$	The variable that holds the generated formal result
...	...

receiving i under any condition and the latter indicates that if each $g \in G$ is satisfied on s, the corresponding transition will be activated without any input.

In order to enable accurate description of input and output in FSM models, several symbols are introduced as shown in Table 1.

With the definition of the FSM language, the knowledge of the tool can be given by first analyzing the application process of the pattern system, which consists of two stages: pattern selection and pattern application. Pattern selection is conducted by guiding the developer through the pattern system from its top categories to a pattern p in the bottom level. And the *explanation* item of p helps confirm that p is able to assist the description of the intended function.

As soon as the decision on selecting p is made, the application of p starts from specifying all the elements in its *constituents* item under the guidance produced from "rules for guidance". The obtained element information is then analyzed in the context of the *solution* item and a formal result is generated by applying the appropriate mapping. If there still exist informal expressions in the formal result, they will be further formalized by applying the involved patterns until reaching a formal expression.

Figure 4 shows the FSM model of the above application process of the pattern system where $s0$ is the initial state and s is one of the accept states.

Paths connecting state $s0$ and $s8$ represent the pattern selection process and the selected pattern is determined on state $s8$. Originating from state $s8$, different paths indicate application processes of different patterns. Due to the sake of space, only the path for the pattern *delete* is drawn as an example. It starts from specifying the first element *obj* in the *constituents* item. The FSM model will then be transferred to the next state according to the data type of the given *obj*, which reflects rule1 in "rules for guidance". After all the four elements are clarified, state $t6$ will be reached and the generation of the formalization result is started. Transition $t6 \rightarrow r0$ indicates one of the generation paths which sets variable $formalExp$ as "$override(\tilde{}obj, v \rightarrow delete(\tilde{}obj(v)))$". Since an informal expression $delete(\tilde{}obj(v))$ is included, the FSM is not terminated on $r0$ and the

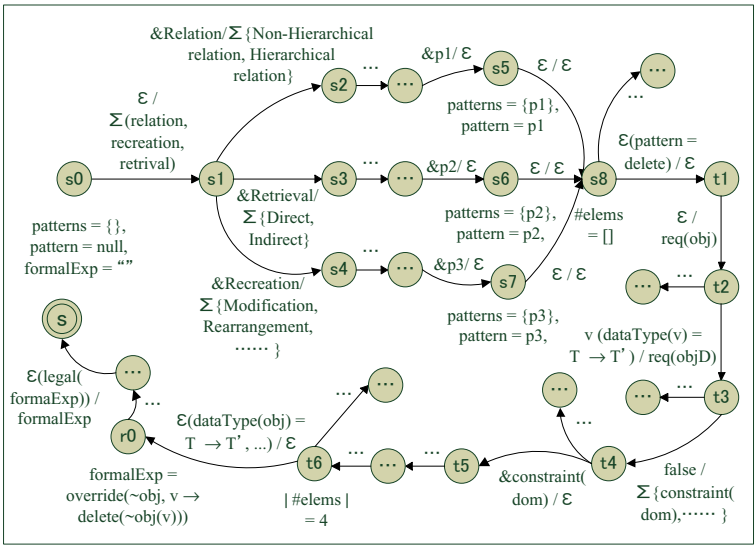

Fig. 4. The FSM modeling the application process of the pattern system

paths originating from $r0$ describe the formalization of the expression. The end state of these paths is the accept state s where $legal(formalExp)$ is satisfied.

Core Engine. A knowledge retrieval algorithm is designed to be implemented by the core engine to utilize the FSM model and determine the behavior of the tool. It is conducted as the following steps:

1. Set the current state cs as $s0$ of the FSM model, and set output as empty.
2. Receive input from the developer if the output requires responses.
3. Analyze the labels in $Acc(cs)$. For the satisfied transition $s \rightarrow s'$ labeled $i(G)/o$, display o to the designer if it is not empty, set the current state as s' and modify variable values according to $\varphi(s')$.
4. If the current state is an accept state, the tool is terminated with a formal expression. Otherwise, repeat step 2-4.

3.3 Tool Implementation

The major concern when implementing the above design is the format for storing the FSM model in the tool. Considering that XML is becoming widely used in industry for its simplicity, it is used to carry the information in the knowledge base so that the knowledge can be better shared by other communities.

As a markup language, XML requires a set of tags to identify data with different meanings. Table 2 shows the XML tags for the FSM model.

In order to demonstrate how the FSM model is represented using the above tags, the previous example FSM written in XML is given in Figure 5.

Table 2. Tags for identifying the elements in the FSM model

tag	the corresponding elements in the FSM model
<state>	states in Q
<transition>	transitions originating from certain state
<input>	input of a transition label
<guard>	guard condition of a transition label
<output>	output of a transition label
<dest>	destination states of transitions
<inf>	value information of variables on states
<para>	variable names
<value>	the value of variables
...	...

```
<state>
  <name>s</name>
  <inf>
   <para>x</para>
   <value>v1</value>
  </inf>
  <inf>
   <para>y</para>
   <value>v2</value>
  </inf>
  <transition>
   <input>a</input>
   <guard></guard>
   <output>b</output>
   <dest>s'</dest>
  </transition>
</state>
```

```
<state>
 <name>s'</name>
 <inf>
  <para>x</para>
  <value>v1+a</value>
 </inf>
 <transition>
  <input>c</input>
  <guard>x>0</guard>
  <output>d</output>
  <dest>s"</dest>
 </transition>
</state>
```

```
<state type = "accept" >
 <name>s"</name>
 <inf>
  <para>y</para>
  <value>v2-c</value>
 </inf>
</state>
```

Fig. 5. The example FSM written in XML

Table 3. The informal explanation of the symbols in the FSM model

symbol	the corresponding informal guidance
$\sum\{s1, ..., sn\}$	Please choose from the following items: 1. s1 ... n.sn
$req(x)$	Please provide a value for x
$\#k$	Please press key k
...	...

Besides, symbols introduced for denoting output in the FSM model are associated with informal explanations to enable automatic production of informal guidance, as shown in Table 3. Let's take the first one as an example. With the informal explanation of symbol $\sum\{s1, ..., sn\}$, output $\sum\{relation, recreation, retrieval\}$ of the transition $s0 \rightarrow s1$ of the FSM model can be automatically transformed into "Please choose from the following items. 1. relation 2. recreation 3. retrieval" when it is exposed to the designer as guidance.

When implementing the tool, the FSM model of the pattern system is stored as knowledge in a XML file based on the pre-defined XML tags. The core engine extracts state and transition information from the file for implementing

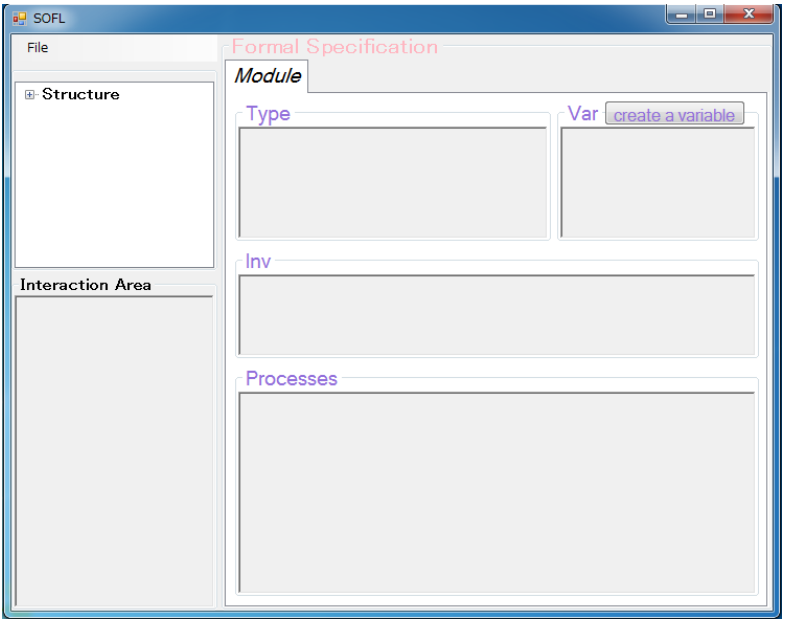

Fig. 6. The interface of the tool

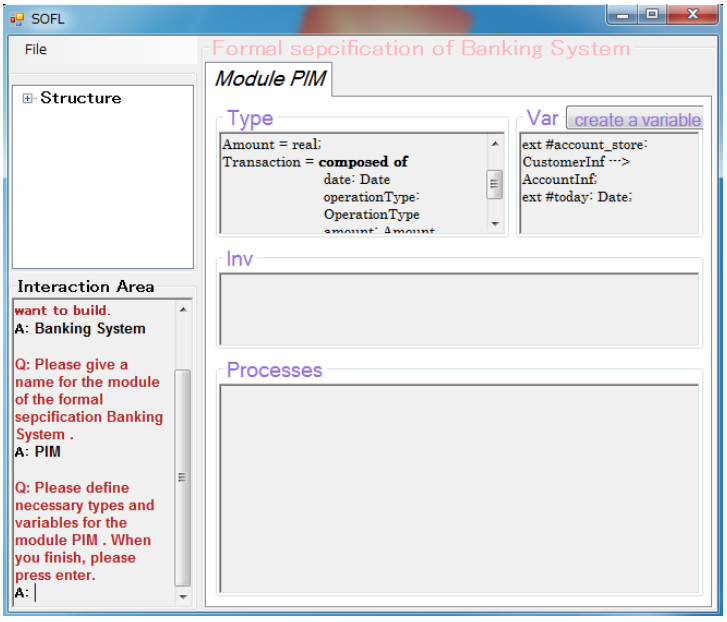

Fig. 7. A snapshot of the tool

the knowledge retrieval algorithm and produces comprehensible guidance with informal explanations attached to symbols in output.

Figure 6 shows the interface of the tool where the top left part gives the architecture of the formal specification under construction, the right half part displays the content of the selected module with different parts shown in different areas. The most important component "Interaction Area" in the bottom left exposes guidance and receives response from the designer.

4 Case Study

A case study on a banking system is presented to illustrate how the tool works. The banking system mainly provides five services to its customers: *deposit*, *withdraw*, *currency transfer* and *private information management*. In the function *private information management*, the customer is able to check his account information including balance and transaction history, and delete unnecessary data from the account. For the sake of space, we take the deletion of the designated transactions as an example.

After creating a new specification for the banking system from the "File" menu, the tool begins to guide the description of each module sequentially, including the declaration of types and variables, and the writing of pre- and post-condition. Assume that a module named *PIM* is created for describing the function *private information management* and all the necessary types and variables are already defined as shown in Figure 7.

In module PIM, process $tranDel$ corresponds to the function of deleting designated transactions from certain account. It takes two inputs: $accountNo$ denoting the account from which the designated transactions are required to be deleted and $delDate$ denoting the date of the transactions to be deleted, and produces an output msg denoting a message for conveying the success of the deletion operation.

The pre-condition of the process is set as true and writing formal expression for the post-condition needs assistance, which activates the core engine to start working on the FSM model stored in the XML file. Running from state $s0$, the tool first captures the informal idea, through interactions, on an abstract level where the category of the intended function is determined. Since the main behavior of process $tranDel$ is a deletion operation, "deleting existing items" will be chosen under guidance, which is reflected by the "Interaction Area" of the tool as shown in Figure 8.

Then the tool helps clarify the details of the intended deletion operations. Leaving state $s8$ for state $t2$ where the first element obj is required to be specified, the core engine generates the corresponding informal requirement "Specify the object you intend to delete data items from." in the Interaction Area. According to the example deletion function, deletion operation is performed on the database that stores the information of all the valid accounts, which is represented by variable $account\_store$ defined in the specification. Thus, variable $account\_store$ should be the object that contains the data items to be deleted

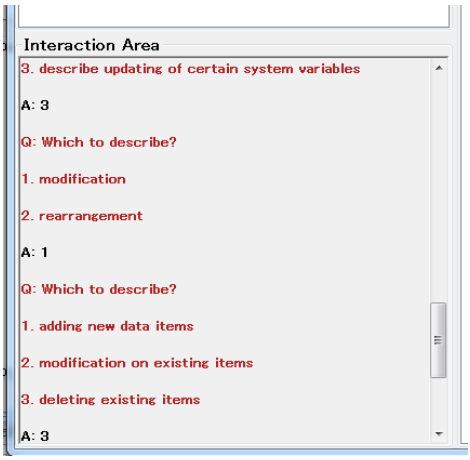

Fig. 8. A snapshot of the Interaction Area

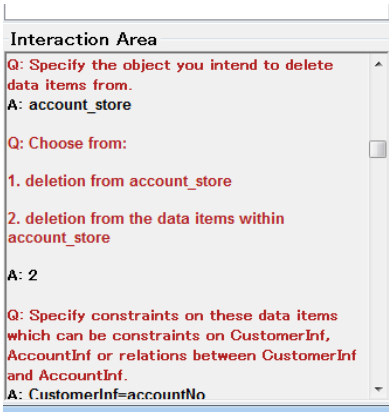

Fig. 9. A snapshot of the Interaction Area

and will be designated as the value of *obj* when provided as the response to the above requirement. With the given value, the core engine analyzes all the transitions originating from state $t2$ and obtains the one whose guard condition is satisfied. Due to the fact that variable *account_store* is of mapping type, transition $t2 \rightarrow t3$ is activated and the corresponding output $req(objD)$ is displayed as "Choose from: 1. deletion from *account_store* 2. deletion from the data items within *account_store*". Once decision is made, the core engine continues traversing the FSM model in the knowledge base and leads the interaction with the designer accordingly, until reaching state $t6$ where all the necessary information for describing the example deletion function is achieved. Part of the above process is shown in Figure 9.

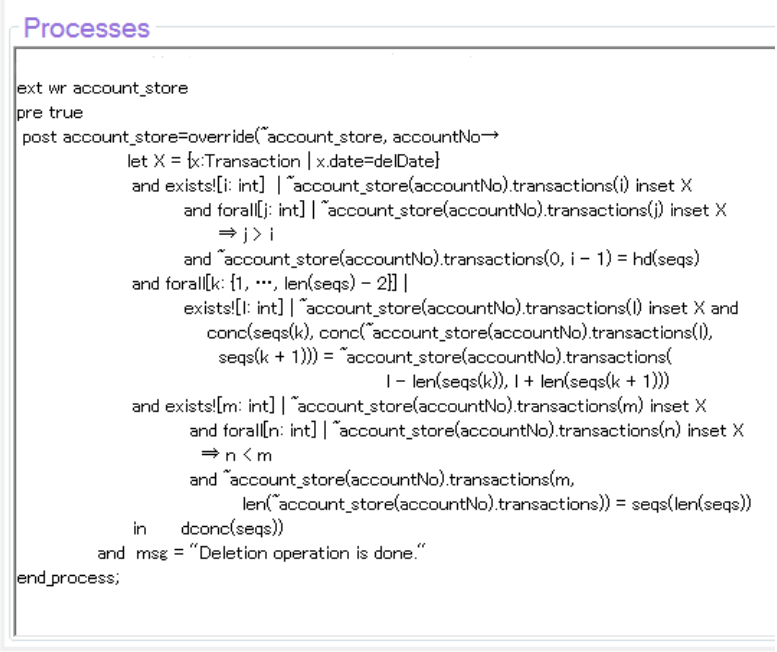

Fig. 10. A snapshot of the Processes Area

Transition $t6 \rightarrow r0$ will then be activated since its guard conditions are all satisfied by the given element values. On the destination state $r0$, a formalization result is designated to variable $formalExp$. It can be seen from the FSM model, state $r0$ is not an accept state and the tool will keep on guiding the designer to formalize the informal expressions in the formalization result. Finally, the core engine terminates on the accept state s with a formal expression automatically generated and designated to the post-condition of process $tranDel$ shown in Figure 10. Within the context of SOFL language, the generated formal expression correctly reflect the retrieved informal idea.

Such a case study shows that the tool is able to remind the necessary aspects in describing certain kinds of functions, capture the informal ideas of the intended functions and organize the obtained information into formal expressions, through interactions. Since the interactions are done in natural language and the tool handles all the tasks involving formal notations, designers can be free from insignificant details and focus on function design.

5 Conclusion

This paper describes a tool for supporting the process of formalizing software requirements. Our main goal is to prevent practitioners from overwhelming tasks

on formal notation details. Instead of automating the whole process, human beings are expected to help make critical decisions since designing system behaviors is an intelligent activity.

Although the case study presented in this paper shows potential strength of the tool, it is relatively small and an empirical one is needed. We intend to invite industrial practitioners to use the tool in real practice so that the feasibility of the underlying theory can be evaluated and the usability of the tool can be improved.

To facilitate the overall construction process, methods for supporting other aspects of formal specifications need to be considered, such as declaration of types and variables, and architecture design. Besides, due to the fact that the performance of the tool largely depends on the knowledge base, how to efficiently update it is a major concern. We intend to explore a self-learning mechanism in the future which enables the knowledge base to be updated through interactions.

Acknowledgement. This work is partly supported by the SCAT Foundation and Hosei University. It is also partly supported under the National Program on Key Basic Research Project(973 Program) Grant No. 2010CB328102; and NSFC No 61133001.

References

1. Raymond Abrial, J.: Formal methods: Theory becoming practice. Journal of Universal Computer Science, 619–628 (2007)
2. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Comput. Surv. 41, 19:1–19:36 (2009)
3. Almeida, J.: An overview of formal methods tools and techniques. Springer (2011)
4. Wang, X., Liu, S., Miao, H.: A pattern system to support refining informal ideas into formal expressions. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 662–677. Springer, Heidelberg (2010)
5. Liu, S.: Formal Engineering for Industrial Software Development. Springer (2004)
6. Liu, S., Offutt, A., Ho-Stuart, C., Sun, Y., Ohba, M.: SOFL: A formal engineering methodology for industrial applications. IEEE Transactions on Software Engineering 24, 24–45 (1998)
7. Liu, S.: Formal engineering for industrial software development – an introduction to the SOFL specification language and method. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 7–8. Springer, Heidelberg (2004)
8. Liu, S.: Integrating top-down and scenario-based methods for constructing software specifications. Inf. Softw. Technol. 51, 1565–1572 (2009)
9. Ding, J., Mo, L., He, X.: An approach for specification construction using property-preserving refinement patterns. In: Proceedings of the 2008 ACM Symposium on Applied Computing, SAC 2008, pp. 797–803. ACM, New York (2008)
10. Stepney, S., Polack, F., Toyn, I.: An outline pattern language for Z: Five illustrations and two tables. In: Bert, D., Bowen, J.P., King, S., Waldén, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 2–19. Springer, Heidelberg (2003)
11. Vadera, S., Meziane, F.: From English to formal specifications. The Computer Journal 37, 753–763 (1994)
12. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, pp. 372–381. ACM, New York (2005)

Abstract Model Checking with SOFL Hierarchy*

Cong Tian[1], Shaoying Liu[2], and Zhenhua Duan[1]

[1] ICTT and ISN Lab, Xidian University, Xi'an, China
{ctian,zhhduan}@mail.xidian.edu.cn
[2] Department of Computer Science, Hosei University, Japan
sliu@hosei.ac.jp

Abstract. Based on the underlying control flow graphs of programs, model checking can be applied to software for effective verification. However, state space explosion forms a major bottleneck that blocks the development of software model checking. Undoubtedly, how to achieve proper abstract models of programs is a key problem. In this paper, instead of the traditional abstraction-refinement method, we present a new abstract model checking approach for efficient verification of software in large scale by utilizing SOFL hierarchy. Within this approach, programs are verified from the high-level to low-level structures, and the state space throughout the verification can be effectively controlled.

Keywords: Model checking, abstraction, SOFL, verification, SPIN.

1 Introduction

Model checking [1–3] is proved to be a successful technology for the verification of hardware. It works, however, on only finite state machines, and most software systems have infinitely many states. Therefore, kinds of techniques are developed for obtaining finite models of software. Even though, the state space of the model can be exponentially larger than the description of the program. This problem, known as state space explosion, is one of the biggest stumbling blocks to the practical application of model checking in the verification of large scale software. How to control state space has therefore been a major direction of research in software model checking [4–7].

Currently, several approaches such as abstraction [8–11], program slicing [12], partial order reduction [13], symbolic [14] and bound [15] techniques, etc., are applied to model checking to reduce the state space for efficient verification. Among the techniques, abstraction is certainly the most important one in software model checking since it can obtain finite models from infinite state space. Traditional abstraction technique preserves all the behaviors of the concrete system but may introduce behaviors that are not presented originally. Thus, if a property is satisfied in the abstract model, it will still be satisfied in the concrete model. But if a property is unsatisfiable in the abstract model, it may still be satisfied in the concrete model, and none of the behaviors that violate the

* This work is supported in part by Hosei University HIF Fellowship, Okawa Foundation, NSFC Grant (No. 61003078, 61272117, 61133001 and 60910004), 973 Program of China Grant (No. 2010CB328102), and ISN Lab Grant No. ISN1102001.

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 71–86, 2013.

property in the abstract model can be reproduced in the concrete model. In case a spurious counterexample is found, the abstraction should be refined in order to eliminate the spurious behaviors. This process is repeated until either a real counterexample is found or the abstract model satisfies the property. Throughout the abstraction-refinement loop, how to check whether or not a counterexample is a real one, and how to achieve a smallest (coarsest) refined model are both difficult problems [9, 16, 17]. In addition, infinite iterations of abstraction-refinement may occur in case of infinite systems. Extremely, there may exist a counterexample that cannot be judged to be spurious or real.

In this paper, we present a new abstract model checking approach for efficient verification of software in large scale by utilizing SOFL hierarchy. SOFL, standing for Structured Object-oriented Formal Language [18–20], is both a specification language and a method that integrates formal methods with commonly used techniques for requirements analysis, design, and verification in software engineering. A system specification written in SOFL is a set of related modules in a hierarchical fashion. Each module is a functional abstraction represented by a Condition Data Flow Diagram (CDFD) where each vertex denotes a process (an operation, or a set of exclusive operations) and each edge indicates a data flow. Within this approach, programs are verified from the high-level (abstract model) to low-level (partially refined model) structures, and the state space throughout the verification can be effectively controlled.

To this end, how a finite state model can be extracted from the underlying structure, CDFD, of a SOFL specification is investigated. Then how consistency properties such as feasibility, composability, invariant-conformance consistency and decompositional consistency can be verified is discussed. Assuming the consistency of the specification itself, whether the specification can satisfy the desired properties in the requirement is verified. We also show how the verification can be implemented using the model checker SPIN [21]. The main contributions of this paper are in two folds: (1) A new kind of abstract model checking is proposed where the abstraction-refinement loop is replaced by a model checking process in hierarchy style. (2) An automatic approach to feasibility and composability checking of pre- and post-condition based specification is proposed.

The reminder of this paper is structured as follows. Section 2 briefly introduces the basic concepts, like hierarchy, processes, data stores and control structures in SOFL. In Section 3, how a SOFL specification can be transformed to a Kripke structure is presented with coarse graphs being the intermediary. In sequel, how the Kripke structure can be represented as a PROMELA model where some inconsistency traps are inserted into is discussed in Section 4. Section 5 is concerned with the verification of consistency of specifications as well as the desired properties in the requirements. In Section 6, the framework of SOFL hierarchy based abstract model checking is drawn. Finally, the conclusion and future research directions are pointed out in Section 7.

2 SOFL Hierarchy

Generally speaking, a system specification written in SOFL is a set of related modules in a hierarchical fashion. Each module is a functional abstraction represented by a CDFD where each vertex denotes a process (or an operation) and each edge indicates a data

flow. As illustrated in Fig.1, the high level specification is portrayed by the CDFD-1, and a refined specification produced by decomposing processes *P* and *Q* is described by CDFD-2 and CDFD-3, respectively. In what follows, we briefly present processes, data stores and some structures in CDFDs.

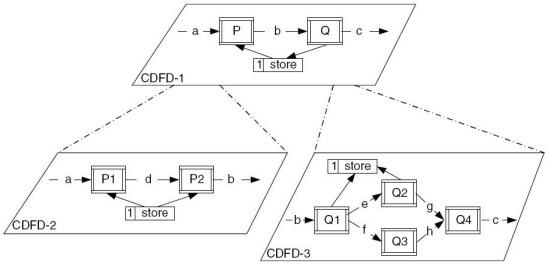

Fig. 1. An illustration of SOFL Hierarchy

2.1 Processes

A process performs an action, task, or operation that takes input and produces output. To model the variety of operations, a process can take several different forms: single port and multiple ports. A process is composed of five parts: *name*, *input port*, *output port*, *pre-condition* and *post-condition*. Fig. 2 (1) shows a single port process. The name A of the process is given in the center of the box. The input port is denoted by the narrow rectangle on the left part of the box, which receives the input data flows x_1, ..., x_n. The output port is given on the right part of the box, similar to the input port, to connect to the output data flows y_1, ..., y_m. The upper part of the box, a narrow rectangle, denotes the precondition, while the lower part of the box represents the postcondition. Fig. 2 (2) depicts a process, named B, with multiple ports. Each $\mathsf{x_i}$ ($i = 1, ..., n$) or $\mathsf{y_j}$ ($j = 1, ..., m$) denotes a group of data flows. The short horizontal lines between the input ports denote the exclusive relation between the groups of data flows in the sense that only one of them can be consumed in producing the output data flow. Similarly, the short horizontal lines between the output ports related to $\mathsf{y_1}$, ..., $\mathsf{y_m}$ denote the exclusive relation among $\mathsf{y_1}$, ..., $\mathsf{y_m}$. Note that, which of $\mathsf{y_1}$, ..., $\mathsf{y_m}$ is generated can be nondeterministic; but it can also be deterministic, depending on how process B is formally specified. For convenience, a process with *m*, $m \geq 1$, input ports and *n*, $n \geq 1$, output ports is called a *m-n* process.

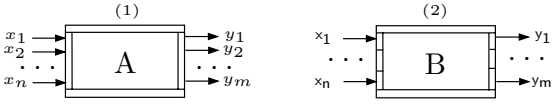

Fig. 2. Processes

2.2 Data Stores and Structures

In SOFL, a data store (store for short) is defined as a variable that holds data in rest. In contrast to data flows, stores do not actively transmit data to any process; rather they hold data that is always ready to supply to any process when requested. A store has a name and number for reference by people who are involved in the building of the specification. There are two ways to connect a store to a process, each denoting a different kind of access to the data in the store by the process: read and write. Fig.3 illustrates the different connections between processes and stores. The connection between store s_1 and process A on the left hand side represents a read from the store by the process during its execution. Note that s_1 stays unchanged before and after the execution of the process. The connection between store s_2 and process B given on the right hand side represents a writing to or updating s_2 by B. This does not exclude the case of reading data from s_2, but must ensure that writing to the store is definitely involved. A writing to the store may include the cases of updating part of the data of s_2 or completely replacing the current data of s_2 with new data.

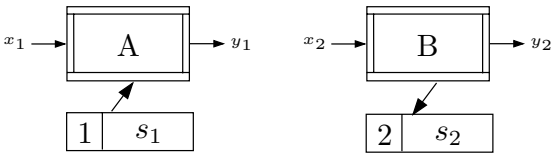

Fig. 3. Connections between processes and stores

To facilitate the usage of SOFL specifications in practice, several other structures, including single condition structure, binary condition structure, multiple conditions structure, non-determinism structure, and broadcasting structure, ect., are provided in SOFL. Considering the facts that these structures can be easily modeled by processes and data flows, and that they are not directly used in this paper, we omit the corresponding introduction to save space.

3 From SOFL Specifications to Kripke Structures

For the purpose of model checking, this section focuses on how to extract a finite state model from the underlying structure, CDFD, of a specification written in SOFL. To this end, we first transform a CDFD to a state-based graph structure with parallel by decomposing processes as operations. Further, a Kripke structure [23] is obtained by replacing parallel with interleaving, and then decomposing states according to Disjunctive Normal Forms (DNF) of predicates obtained from the pre- and post-conditions.

3.1 Detachment of Processes

Actually, a process in SOFL can be seen as a capsulation of a set of exclusive operations. This relation is formally expressed below.

We adopt the definition of operations presented in [18] and modify it slightly.

Definition 1. An operation OP is a four tuple, $OP = (OP_{IV}, OP_{OV}, OP_{Pre}, OP_{Post})$, where OP is the name, OP_{IV} the set of all the input variables, OP_{OV} the set of all the output variables, OP_{Pre} the pre-condition, and OP_{Post} the post-condition of the operation. □

For convenience, we also represent a process as a tuple.

Definition 2. A *m-n* process P is a four tuple, $P = (IV, OV, Pre, Post)$, where $IV = \{IV_1, IV_2, ..., IV_m\}$ is the set of input variable groups with respect to the m input ports; similarly, $OV = \{OV_1, OV_2, ..., OV_n\}$ is the set of output variable groups with respect to the n output ports; *Pre* and *Post* are the pre- and post-conditions of the process, respectively. □

Based on the above definitions, a *m-n* process, $P = (IV, OV, Pre, Post)$, can be further represented as a set of operations: $P = \{\mathsf{P\_i\_j} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$. For each i and j, $\mathsf{P\_i\_j} = (\mathsf{P\_i\_j}_{IV}, \mathsf{P\_i\_j}_{OV}, \mathsf{P\_i\_j}_{Pre}, \mathsf{P\_i\_j}_{Post})$, where $\mathsf{P\_i\_j}_{IV} = IV_i$, $\mathsf{P\_i\_j}_{OV} = OV_j$, $\mathsf{P\_i\_j}_{Pre} = Pre$, and $\mathsf{P\_i\_j}_{Post} = Post$. Note that the name, P_i_j, of the operation indicates that the operation is obtained by consuming the data flows received by the *i*th input port and producing data flows to the *j*th output port of process P. Apparently, a *m-n* process can be detached as $m \times n$ operations. Especially, when P is a single port process, only one operation is obtained.

Based on the relationship between operations and processes, as shown in Fig. 4, a *m-n* process P can be intuitively detached as the nondeterministic mode of all the operations contained in P. Accordingly, by detaching all the processes as operations, a CDFD can be transformed into a state-based graph. In such a graph, each node is named by P_i_n (or P_o_n), meaning that this node is obtained from the *n*th input (or output) of process P. Further, the labels of node P_i_n (or P_o_n) are *Pre* and IV_n (or *Post* and OV_n) which are adopted from process $P = (IV, OV, Pre, Post)$, directly.

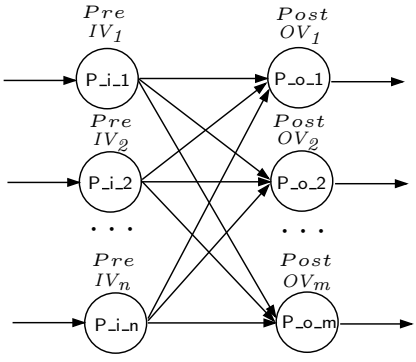

Fig. 4. Detachment of a process

3.2 Data Stores

Data stores in SOFL can be treated as 'rd' (readable) or 'wr' (writable) external variables. Specifically, a 'rd' external variable of operation *OP* provides an unmodifiable input value to the operation, while a 'wr' external variable provides both input and output values to the operation. Based on this, for a 'rd' external variable, it is included in OP_{IV} as an input variable, and for a 'wr' external variable, we add it into both OP_{IV} and OP_{OV} as input and output variables of the operation *OP*. Thus, it is unnecessary to consider the operations on data stores individually. To center on the essence of the verification approach, all the variables involved are restricted to numeric types.

3.3 Elimination of Parallel

In the graph structure obtained by detaching processes, parallel components may exist since processes in CDFDs may execute in parallel. For instance, in the CDFD shown in Fig. 5 (1), processes *B* and *C* may execute in parallel. Correspondingly, in the graph structure illustrated in Fig. 5 (2) obtained by detaching processes, the two branches [B_i_1, B_o_1] and [C_i_1, C_o_1] departing from state A_o_1 can proceed in parallel. However, in finite state models, such as automata, Kripke systems and transition systems, true concurrency is not permitted. Instead, interleaving is used to describe the parallel executions.

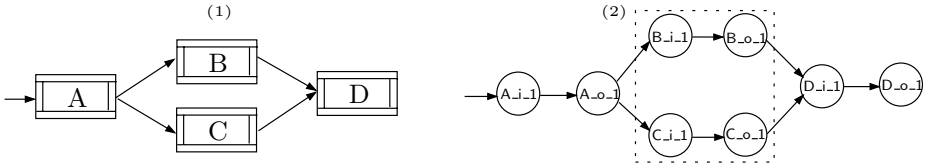

Fig. 5. CDFD with parallel

To replace a parallel component with interleaving, all the parallel branches in the parallel component are shuffled. For example, as illustrated in Fig. 6, the parallel component in Fig. 5 (2) is replaced with interleaving. Accordingly, with this method, all the parallel components in the graph obtained from a CDFD can be eliminated. For clarity, the graph obtained from a CDFD by detaching processes and then eliminating parallel is called a coarse graph.

3.4 Construction of Kripke Structures

After eliminating parallel components, the underlying structure of a finite state model is built. However, each state in the established coarse graph is labeled with a predicate logic formula obtained from the pre- or post-conditions in the specification as well as a set of input or output variables. Now we focuses on how to further transform the coarse graph to a Kripke structure where each state is labeled with a set of atomic predicates.

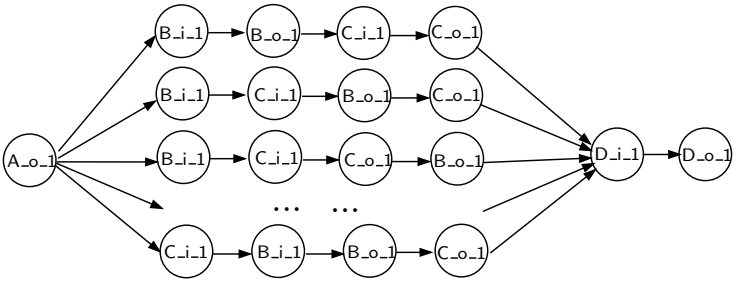

Fig. 6. Replacement of parallel with interleaving

For the predicate F labeled on a state s in the coarse graph, we first transform F to DNF, $F = \bigvee_i \bigwedge_j Q_{ij}$, where each Q_{ij} is an atomic predicate. Let $Var(Q_{ij})$ be the set of variables appearing in Q_{ij}, V the set of variables labeled on state s, and V'_1, ..., V'_n the set of variables labeled on state s'_1, ..., s'_n, respectively. Q_{ij} is eliminated from of F if one of the following two conditions holds:

- s is obtained from an input port of a process in the original CDFD, and $Var(Q_{ij}) \cap V = \emptyset$;
- s is obtained from an output port of a process, say P, s'_1, ..., s'_n are the corresponding states formed from the input ports of P, and $Var(Q_{ij}) \cap (V \cup V'_1 \cup ... \cup V'_n) = \emptyset$.

For clarity, the DNF achieved by eliminating redundant atomic predicates is denoted as F'. Subsequently, s is decomposed into several states with respect to the conjunction clauses in F'. We use an example to explicitly illustrate the essence of the transformation. For state s_1 in the coarse graph shown in Fig.7 (1) with predicate F being the label, we first transform formula F to disjunctive normal form, and then eliminate the redundant atomic predicates. Without loss of generality, we assume $F' = Q_1 \wedge Q_2 \vee \neg Q_1 \wedge Q_3 \vee \neg Q_2 \wedge Q_3$. Accordingly, as illustrated in Fig.7 (2), state s_1 is decomposed into three new states with the labels being $\{Q_1, Q_2\}$, $\{\neg Q_1, Q_3\}$ and $\{\neg Q_2, Q_3\}$, respectively. Meanwhile, the transitions related to the original state s_1 are

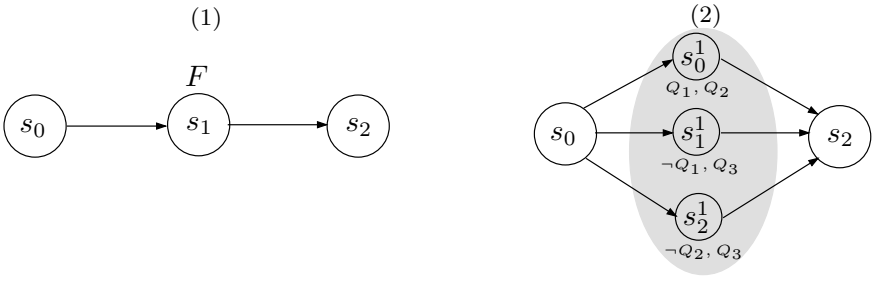

Fig. 7. From coarse graphs to Kripke structures

duplicated and put onto the new created states. With this method, a Kripke structure can be obtained through the coarse graph of a specification.

4 Establishment of PROMELA Models

To detect the inconsistency of a specification, some traps are inserted into the obtained Kripke structure, and a transition system is established.

Recall that in the Kripke structure obtained in the previous section, a node s is labeled with a set S_p of atomic predicates and a set S_v of variables (the set of input or output variables of the corresponding input or output port of the original process). In fact, if s is obtained from an input port, all the variables occurring in S_p are contained in S_v, whereas, there may exist some predicates $P \in S_p$ such that none of the variables occurring in P are contained in S_v, if s is obtained from an output port. In the latter case, according to whether there exist variables occurring in both P and S_v, predicates in S_p are classified into guard conditions and defining conditions. The formal definition is presented below.

Definition 3. In a Kripke structure, for a state s, obtained from an output port, with labels S_p and S_v,

- $P \in S_p$ is called a guard condition if $Var(P) \cap S_v = \emptyset$;
- $P \in S_p$ is called a defining condition if $Var(P) \cap S_v \neq \emptyset$. □

Accordingly, the set S_p of atomic predicates can be separated into $S_p = S_{cp} \cup S_{dp}$, where S_{cp} is the set of guard conditions and S_{dp} the set of defining conditions. Intuitively, a guard condition does not change the value of any variables, while a defining condition redefines (or updates) the value of some variables. Therefore, for a state s obtained from an input port, all the predicates in S_p are guard conditions since pre-conditions do not change the value of any variables.

By treating a state as some operations on variables and a transition as the guard condition of the subsequent operations, the Kripke structure can be represented as a transition system. Specifically, for a node obtained from an input port, say node P_i_1 in Fig. 8 (1), S_p is relabeled on the edge incoming to node P_i_1 as shown in P_i_1 in Fig. 8 (2); while for a node obtained from an output port, for example node P_o_1 in

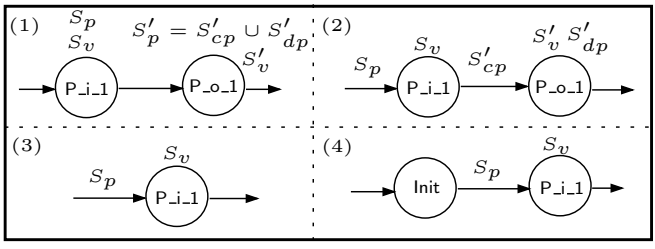

Fig. 8. From Kripke structure to transition system

Fig. 8 (1), the guard conditions S'_{cp} are moved to the edge incoming to node P_o_1 as depicted in Fig. 8 (2). Further, for an initial node, like P_i_1 in Fig. 8 (3), a node Init denoting the initialization of variables is added as the previous node of P_i_1 as illustrated in Fig. 8 (4).

To facilitate the consistency checking, we further add *infeasibility* and *incomposability* traps into the transition systems. As depicted in Fig. 9 (1), when none of the guard conditions in P_o_1, ..., P_o_m can be satisfied, a transition from node P_i_1 to an extra node !Fea, meaning *infeasible*, is added; In contrast, as shown in Fig. 9 (2), when none of the guard conditions in Q_i_1, ..., Q_i_m are satisfied, a transition from node P_o_1 to an extra node !Com, meaning *incomposable*, is included.

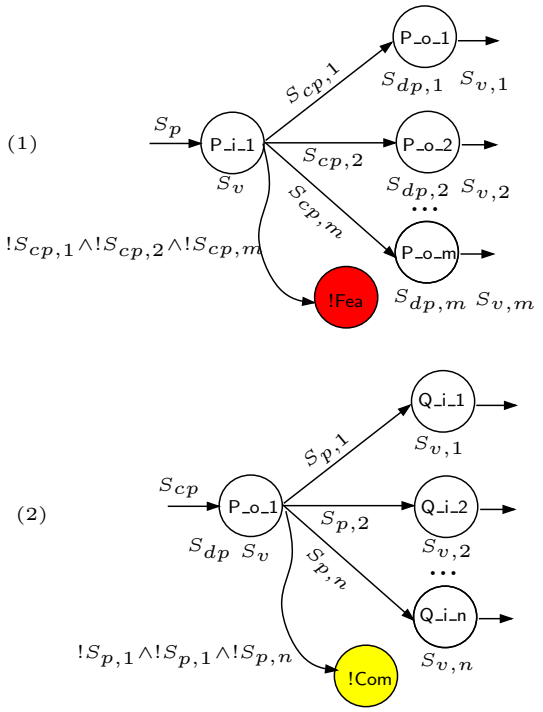

Fig. 9. Transition system with infeasibility or incomposability traps

Subsequently, to implement the verification within the model checker SPIN, the transition system is further expressed in the syntax of PROMELA language which can be accepted by SPIN. The translation from a transition system to a PROMELA model is simple:

- treating each state in the transition system as a *progress state label* in PROMELA;
- expressing the transition relations from a state with a *selection structure* in PROMELA.

Table. 1 shows the outline of PROMELA model of a SOFL specification.

5 Verification of SOFL Specifications

Generally speaking, the properties in a SOFL specification can be categorized into two folds: (1) consistency of the specification itself, e.g. feasibility and composability; (2) satisfaction of the functional requirement, i.e. whether the specification precisely reflects the critical properties, such as safety and liveness, in the requirement.

5.1 Consistency

In literature [20], for the purpose of specification inspection, consistency properties are classified into four categories: feasibility, composability, invariant-conformance consistency, and decompositional consistency. In the remainder of this part, we concentrate on how to manage these four kinds of consistency properties with model checking.

Feasibility. The feasibility of a process means that for any input meeting its pre-condition, there exists an output that satisfies its post-condition, i.e. the consistency between the pre- and post-conditions of a process. Correspondingly, in the transition system, as illustrated in Fig. 10, if node !Fea is reachable from node P_i_1, we say that output port 1 of process *P* is infeasible. Accordingly, if node !Fea is reachable from all of the nodes in P_i_1, ..., P_i_n, we can declare that process *P* is infeasible.

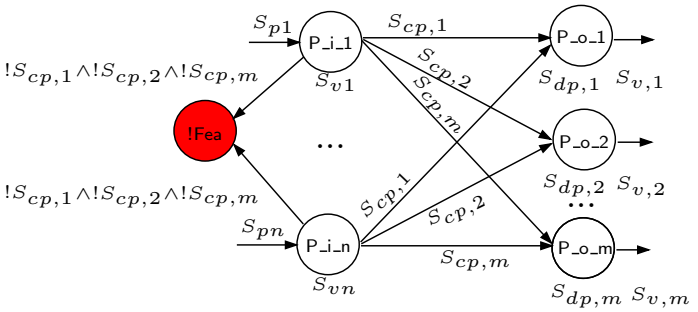

Fig. 10. Feasibility checking

To check the feasibility of a CDFD, we use temporal logic formula $\Box(inFea\ !=\ true)$ to describe that always state !Fea is unreachable. When implemented using SPIN, in the case of no errors being reported, the specification is feasible. Otherwise, some errors are reported, and each time a counter example is provided by the simulator in SPIN. Based on each counter example, an infeasible input port of some process can be detected. For instance, if a counter-example < A_i_1, A_o_1, B_i_1, B_o_1, C_i_1, !Fea > is reported as shown in Fig.11, we say that input port 1 of process *C* is infeasible. By analyzing all the counter-examples, if all the input ports of a process, say *P*, is infeasible, it determines that process *P* is infeasible.

Table 1. Outline of the PROMELA model

```
active proctype CDFD-PROMELA( )
{
type x1; ... type xn;
    /*Declaration and initialization of variables involved in the CDFD*/
bool inFea=0; bool inCom=0;
               /* Boolean variables for Feasibility and composability*/
Init:                                                        /* Init state*/
 if                                          /* Transitions from Init state */
 :: Sp,1 goto P_i_1
 :: ...
 :: Sp,m goto P_i_m
 :: !Sp,1∧!Sp,m goto !Com                          /* Go to !Com state */
 :: ...
 :: Sq,1 goto Q_i_1
 :: ...
 :: Sq,n goto Q_i_n
 :: !Sq,1 ∧ ...∧!Sq,n goto !Com                    /* Go to !Com state */
 fi;
 ... ...
P_i_1:                                                      /* P_i_1 state*/
 if                                        /* Transitions from P_i_1 state */
 :: Scp,1 goto P_o_1
 :: ...
 :: Scq,n goto P_o_n
 :: !Scp,1 ∧ ...∧!Scp,n goto !Com                  /* Go to !Com state */
 fi;
 ... ...
P_o_1:                                                    /* P_o_1 state */
 if                                       /* Transitions from P_o_1 state */
 :: Sr,1 goto R_i_1
 :: ...
 :: Sr,m goto R_i_m
 :: !Sr,1 ∧ ...∧!Sr,m goto !Fea                    /* Go to !Fea state */
 fi;
 ... ...
!Inf:                                                       /* !Inf state*/
     inFea=1; skip;
!Com:                                                      /* !Com state*/
     inCom=1; skip;
}
```

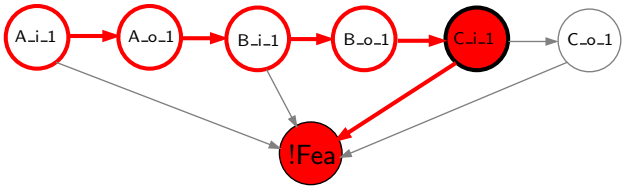

Fig. 11. A counterexample

Composability. Composability is intended to ensure that the pre-condition of a process in a specification is always satisfied by its input data. The pre-condition of a process defines the responsibility for the operational environment that supplies the input data. To ensure this property, the pre-condition of the process and the post-conditions of all the preceding processes must be consistent, because the preceding processes produce the input data flows for the process. Correspondingly, in the transition system, as illustrated in Fig. 12, if node !Com is reachable from node P_o_1 via transition $!S_{p1,1} \wedge !S_{p1,2} \wedge !S_{p1,m}$, we say input port 1 of process Q is incomposable. Accordingly, if all of the input ports of process Q is incomposable, we can declare that process Q is incomposable.

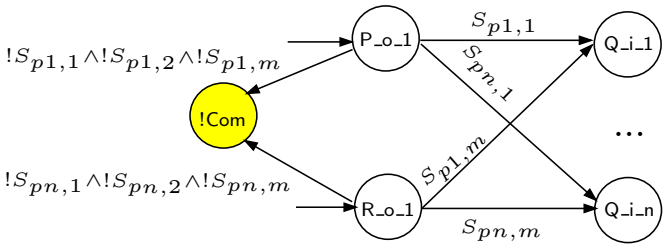

Fig. 12. Composability checking

To check the composability of a CDFD, we use temporal logic formula $\Box(inCom\ != true)$ to describe that always state !Com is unreachable. When implemented using SPIN, in the case of no errors being reported, the specification is composable. Otherwise, some errors are reported, and each time a counter example is provided by the simulator in SPIN. Based on each counter example, an incomposable input port of some process can be detected. For instance, if a counter-example < A_i_1, A_o_1, B_i_1, B_o_1, !Com > is reported as shown in Fig.13, we say that input port 1 of process C is incomposable. By analyzing all the counter-examples, if all the input ports of a process, say P, is incomposable, it determines that process P is incomposable.

Invariant-Conformance Consistency. The invariant-conformance consistency requires that each invariant cannot be violated by the pre- and post-conditions of the related processes. That is, when the pre- or post-condition of a related process evaluates to true, the invariant must hold. This can be directly verified via model checking

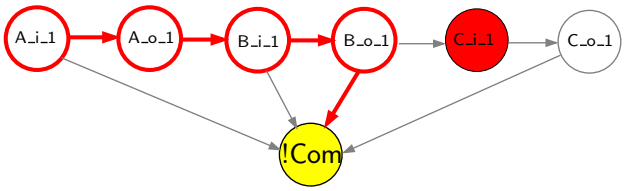

Fig. 13. A counterexample

by expressing the invariant with a temporal logic formula. Here we use an example to show how to express an invariant in a temporal logic formula. Let x be an integer variable ranging over $[-255, 255]$. Thus, throughout the execution, it is always required that $-255 \leq x \leq 255$. This can be expressed by temporal logic formula: $\Box((x \leq 255) \wedge (x \geq -255))$. When implemented using a model checker, anytime, if the invariant is violated, a counterexample is provided to show where it occurs.

Decompositional Consistency. Decompositional consistency is a property concerned with a process and its decomposition. A process is consistent with its decomposition if, and only if, for any input satisfying the pre-condition of the process, the output produced by the decomposition satisfies the post-condition of the process. Without loss of generality, we show how the decompositional consistency can be verified through a simple example. The CDFD depicted in Fig.14 (2) is a decomposition of the process in Fig.14

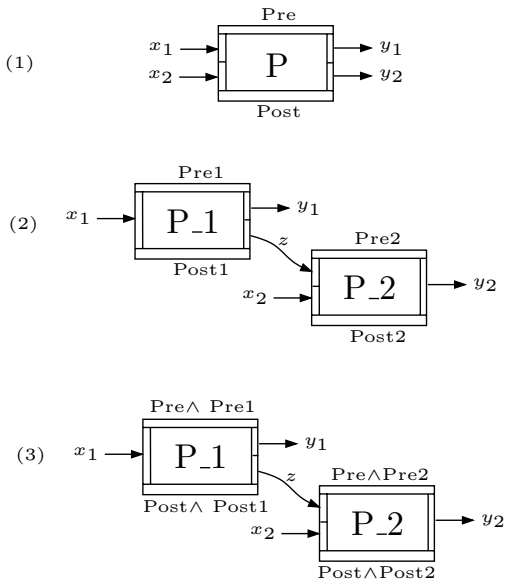

Fig. 14. Kripke structure with decompositional consistency information

(1). The decompositional consistency can be checked via checking the composability of process P_1, as well as the feasibility of processes P_1 and P_2 in Fig.14 (3). The pre-condition of process P_1 (or P_2) in Fig.14 (3) is the conjunction of the pre-conditions of process P and P_1 (or P_2) in Fig.14 (1) and (2), respectively; the post-conditions of processes P_1 (or P_2) in Fig.14 (3) are the conjunction of the post-conditions of process P in Fig.14 (1) and process P_1 (or P_2) in Fig.14 (2), respectively.

5.2 Critical Properties in Functional Requirements

Assuming the consistency of a specification, it is still hard to make sure that the specification correctly reflects the functional requirements of the users. In particular, it is important that whether the significant properties are satisfied in the specification.

The requirements that designers wish to impose on systems fall basically into two categories: (1) safety properties state that "something bad never happens"; (2) liveness properties express that "something good will eventually happen". For a safety property, it is usually described with a temporal logic [22] formula, $\Box\neg p$, where p is a predicate indicating something bad. Whereas, for a liveness property, formula $\Diamond q$ is used to specify that q, something good, will eventually occur. In what follows, take the ATM system (in Japan) for example, we show how to check the satisfaction of the critical properties in users' requirements.

One important property in ATM system is: *No overdraw is allowed*. To express this property in temporal logic formulas, several useful variables are defined first.

```
int  account_balance;   /* The-balance of the account*/
int  withdraw_amount;   /* Amount to be withdrawn*/
bool withdrawn = 0;     /* Requested amount has not been withdrawn*/
```

Accordingly, the property can be specified with the following temporal logic formulas.
Safety: Anytime, if the *withdraw_amount* requested is larger than the current balance, the transaction will never be done.

$$\Box((withdraw\_amount > account\_balance) \rightarrow (withdrawn == 0))$$

Liveness: Anytime, if the *withdraw_amount* requested is less or equal to the current balance, the transaction will eventually be done.

$$\Box((withdraw\_amount <= account\_balance) \rightarrow \Diamond(withdrawn == 1))$$

Providing the PROMELA model and the above desired properties, SPIN can be used to check whether the system satisfies the properties, automatically.

6 Abstract Model Checking Framework

Based on the discussions in the previous sections, we present the framework for abstract model checking with SOFL hierarchy. As portrayed in Fig.15, on abstract models (Kripke structures) of each level, feasibility, composability, invariant-conformance consistency as well as critical properties can be checked,independently. While between the two models under refinement (decomposition) relationship, decompositional consistency are checked.

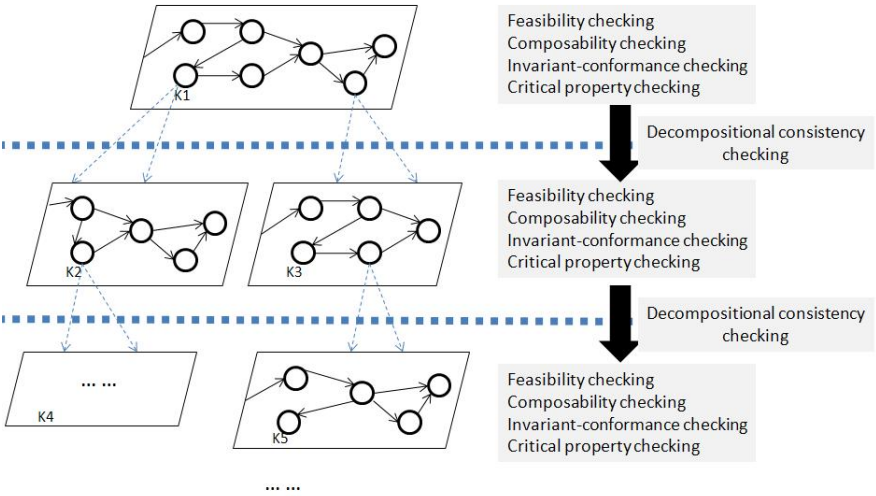

Fig. 15. Abstract model checking framework

7 Conclusion

We present a new abstract model checking approach for efficient verification of software in large scale by utilizing SOFL hierarchy. Within this approach, programs are verified from the high-level to low-level structures, and the state space throughout the verification can be effectively controlled.

In our experience, the method works well for all the examples we adopted currently. However, it still needs to be evaluated via big examples with industrial scale. To do so, supporting tools will be developed in the near future for automatically establishing PROMELA model from a specification written in SOFL. We will also extend the approach to deal with data with complex data structures, such as set, sequence, and composite types.

References

1. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
2. Quielle, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
3. Clarke, E.M., Grumber, O., Peled, D.: Model Checking. MIT Press (2000)
4. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. 41(4) (2009)
5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Software Tools for Technology Transfer 9(5-6), 505–525 (2007)

6. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, L.S., Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: ICSE 2000: Software Engineering, pp. 439–448 (2000)
7. Havelund, K., Pressburger, T.: Model checking Java programs using Java Pathfinder. Software Tools for Technology Transfer (STTT) 2(4), 72–84 (2000)
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
9. Clarke, E.M., Gupta, A., Strichman, O.: SAT Based Counterexample-Guided Abstraction-Refinement. IEEE Trans. Computer Aided Design 23(7), 1113–1123 (2004)
10. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM 50(5), 752–794 (2003)
11. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 443–454. Springer, Heidelberg (1999)
12. Dwyer, M.B., Hatcliff, J.: Slicing Software for Model Construction. In: PEPM 1999, pp. 105–118 (1999)
13. Godefroid, P., Wolper, P.: A Partial Approach to Model Checking. Inf. Comput. 110(2), 305–326 (1994)
14. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. In: LICS 1990, pp. 428–439 (1990)
15. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
16. Tian, C., Duan, Z.: Making Abstraction-Refinement Efficient in Model Checking. In: Fu, B., Du, D.-Z. (eds.) COCOON 2011. LNCS, vol. 6842, pp. 402–413. Springer, Heidelberg (2011)
17. He, F., Song, X., Hung, W.N.N., Gu, M., Sun, J.: Integrating Evolutionary Computation with Abstraction Refinement for Model Checking. IEEE Trans. Computers 59(1), 116–126 (2010)
18. Liu, S., Nagoya, F., Chen, Y., Goya, M., McDermid, J.A.: An Automated Approach to Specification-Based Program Inspection. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 421–434. Springer, Heidelberg (2005)
19. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer, Berlin, ISBN 3-540-20602-7
20. Liu, S., McDermid, J.A., Chen, Y.: A Rigorous Method for Inspection of Model-Based Formal Specifications. IEEE Transactions on Reliability 59(4), 667–684 (2010)
21. Holzmann, G.J.: The Model Checker Spin. IEEE Trans. on Software Engineering 23(5), 279–295 (1997)
22. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th IEEE Symposium on Foundations of Computer Science, pp. 46–67. IEEE, New York (1977)
23. Kripke, S.A.: Semantical analysis of modal logic I: Normal propositional calculi. Z. Math. Logik Grund. Math. 9, 67–96 (1963)
24. D'Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27(7), 1165–1178 (2008), doi:10.1109/TCAD.2008.923410

Model Checking C Programs with MSVL*

Yan Yu[1], Zhenhua Duan[1,**], Cong Tian[1], and Mengfei Yang[2]

[1] ICTT and ISN Lab, Xidian University, Xi'an, 710071, P.R. China
[2] China Academy of Space Technology, Beijing, 100094, P.R. China

Abstract. This paper presents an approach for model checking C programs with MSVL. To do so, we translate C programs into MSVL (modeling simulation and verification language) programs, and specify the desired property by a propositional projection temporal logic (PPTL) formula; then we employ the unified model checking approach to check whether the MSVL program satisfies the PPTL formula. If so, the program is correct; otherwise, a counterexample can be found. The translation algorithm from C to MSVL programs is introduced in details. In addition, an example is given to illustrate how the approach works.

Keywords: Temporal Logic, MSVL, Model Checking, Verification, Translation.

1 Introduction

C is one of the most widely used programming languages for the development of software systems. How to guarantee the correctness and reliability of C programs is a grand challenge to computer scientists and software engineers. In the past four decades, a number of testing techniques and verification approaches have been proposed for testing and verifying C programs with success. In particular, model checking is an automatic approach for verification[1–3] of C programs. With this approach, to verify a C program, an abstract model which describes the behaviors of the program must be extracted from the C program. Further, the property[4–6] to be verified can be specified by a temporal logic[7, 9, 10] (TL) formula. Then, a model checker is employed to check whether or not the model satisfies the property. If the model cannot satisfy the property, a counterexample is provided. As we can see, using model checking for verifying C programs suffers from the process which extracts the model from C programs since this process is not straightforward. On the other hand, with temporal logic programming, such as MSVL (modeling simulation verification language), approach for model checking, the behaviors of a system is described by an MSVL program and the property to be verified is described by a propositional projection temporal logic (PPTL) formula. Then, the unified model checking algorithm can be employed to verify whether or not the MSVL program satisfies the PPTL formula. As a matter of fact, with this approach, to verify C programs we have to rewrite the C program into MSVL programs, this is another burden for programmers. So we are motivated to work out a general translation

* This research is supported by NSFC Grants (No. 61133001, 6091004, 61272117, 61272118, 61003078, and 61202038), 973 Program (No.2010CB328102), and ISN Lab Grant No. ISN1102001.

** Corresponding author.

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 87–103, 2013.

program which can automatically transfer a C program into a MSVL program, so that the model checking process can be automatically and directly conducted based on C programs.

To do so, we first analyze syntax and semantics of the C programs by means of lex and yacc within Paser Generator, and store the information in terms of a specified storage structure; second, we present an algorithm to achieve the translation from C statements to MSVL statements one by one; finally, all of statements in MSVL corresponding to C statements are generated. As a result, whenever a C program is input as a parameter, an equivalent MSVL program is produced. Therefore, to verify a C program, we only need to translate the C program into MSVL program and then verify the MSVL program based on the existing verification techniques.

The contributions of this paper are two-fold: (1) We design and implement a translator in C++ which translates any C program into an equivalent MSVL program. (2) We give an example to show how the translator works and conduct a model checking process to verify the property of the C program automatically.

The rest of this paper is organized as follows. In the next section, the unified model checking approach with MSVL and PPTL is briefly presented. In Section 3, an algorithm is formalized to implement the translation from C programs to MSVL programs. An example is given in Section 4 to show how the translation algorithm works, and how we use the algorithm to achieve our goals of directly verifying a C program. Finally, conclusions are drawn in Section 5.

2 Preliminaries

2.1 Projection Temporal Logic

2.1.1 Syntax

Let Π be a countable set of propositions, and V a countable set of typed static and dynamic variables. $B = \{true, false\}$ represents the boolean domain and D denotes all the data we need including integers, strings, lists, etc. The terms e and formulas p are given by the following grammar:

$$e ::= v \mid \bigcirc e \mid \ominus e \mid f(e_1, \ldots, e_n)$$
$$p ::= \pi \mid e_1 = e_2 \mid P(e_1, \ldots, e_n) \mid \neg p \mid p_1 \wedge p_2 \mid \exists v : p \mid \bigcirc p \mid (p_1, \ldots, p_m)\ \mathit{prj}\ p$$

where $\pi \in \Pi$ is a proposition, and v a dynamic or static variable. In $f(e_1, \ldots, e_n)$ and $P(e_1, \ldots, e_n)$, f is a function and P a predicate. It is assumed that the types of the terms are compatible with those of the arguments of f and P. A formula (term) is called a state formula (term) if it does not contain any temporal operators ($i.e. \bigcirc, \ominus$ and prj), otherwise it is a temporal formula (term).

2.1.2 Semantics

A state s is a pair of assignments (I_v, I_p) where for each variable $v \in V$ defines $s[v] = I_v[v]$, and for each proposition $\pi \in \Pi$ defines $s[\pi] = I_p[\pi]$. $I_v[v]$ is a value in D or nil (undefined), whereas $I_p[\pi] \in B$. An interval $\sigma =< s_0, s_1, \cdots >$ is a non-empty (possibly infinite) sequence of states. The length of σ, denoted by $|\sigma|$, is defined as ω

if σ is infinite; otherwise it is the number of states in σ minus one. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider the set N_0 of non-negative integers and ω, $N_\omega = N_0 \cup \{\omega\}$, and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0, i < \omega$. Moreover, we define $\preceq$ as $\leq -\{(\omega, \omega)\}$. With such a notation, $\sigma_{(i..j)}(0 \leq i \preceq j \leq |\sigma|)$ denotes the sub-interval $< s_i, \ldots, s_j >$ and $\sigma(k)(0 \leq k \preceq |\sigma|)$ denotes $< s_k, ..., s_{|\sigma|} >$. The concatenation of σ with another interval (or empty string) σ' is denoted by $\sigma \cdot \sigma'$. To define the semantics of the projection operator we need an auxiliary operator for intervals. Let $\sigma =< s_0, s_1, \cdots >$ be an interval and $r_1, \ldots, r_h$ be integers $(h \geq 1)$ such that $0 \leq r_1 \leq r_2 \leq \cdots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the interval (called projected interval), $\sigma \downarrow (r_1, \ldots, r_h) =< s_{t_1}, s_{t_2}, \ldots, s_{t_l} >$, where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates. For example,

$$< s_0, s_1, s_2, s_3, s_4 > \downarrow (0, 0, 2, 2, 2, 3) =< s_0, s_2, s_3 >$$

An interpretation for a PTL term or formula is a tuple $I = (\sigma, i, k, j)$, where $\sigma =< s_0, s_1, \cdots >$ is an interval, i and k are non-negative integers, and j is an integer or ω, such that $i \leq k \preceq j \leq |\sigma|$. We use (σ, i, k, j) to mean that a term or formula is interpreted over a subinterval $\sigma_{(i..j)}$ with the current state being s_k. For every term e, the evaluation of e relative to interpretation $I = (\sigma, i, k, j)$ is defined as $I[e]$, by induction on the structure of a term, as shown in Fig.1, where v is a variable and $e_1, \ldots, e_m$ are terms.

$$\mathcal{I}[v] = s_k[v] = I_v^k[v]$$

$$\mathcal{I}[\bigcirc e] = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ nil & \text{otherwise} \end{cases}$$

$$\mathcal{I}[\ominus e] = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ nil & \text{otherwise} \end{cases}$$

$$\mathcal{I}[f(e_1, \ldots, e_m)] = \begin{cases} f(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) & \text{if } \mathcal{I}[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases}$$

Fig. 1. Interpretation of PTL terms

The satisfaction relation, $\models$, for formulas is inductively defined as follows.

1. $\mathcal{I} \models \pi$ if $s_k[\pi] = I_p^k[\pi] =$ true.
2. $\mathcal{I} \models e_1 = e_2$ if $\mathcal{I}[e_1] = \mathcal{I}[e_2]$.
3. $\mathcal{I} \models P(e_1, \ldots, e_m)$ if P is a primitive predicate other than = and, for all h, $1 \leq h \leq m, \mathcal{I}[e_h] \neq nil$ and $P(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) =$ true.
4. $\mathcal{I} \models \neg p$ if $\mathcal{I} \not\models p$.
5. $\mathcal{I} \models p_1 \wedge p_2$ if $\mathcal{I} \models p_1$ and $\mathcal{I} \models p_2$.
6. $\mathcal{I} \models \exists v : p$ if for some interval σ' which has the same length as σ, $(\sigma', i, k, j) \models p$ and the only difference between σ and σ' can be in the values assigned to variable v at k.
7. $\mathcal{I} \models \bigcirc p$ if $k < j$ and $(\sigma, i, k+1, j) \models p$.

8. $\mathcal{I} \models (p_1, \ldots, p_m)\, prj\ q$ if there exist integers $k = r_0 \leq r_1 \leq \ldots \leq r_m \leq j$ such that $(\sigma, i, r_0, r_1) \models p_1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ (for $1 < l \leq m$), and $(\sigma', 0, 0, |\sigma'|) \models q$ for one of the following σ':
   (a) $r_m < j$ and $\sigma' = \sigma\downarrow(r_0, ..., r_m) \cdot \sigma_{(r_{m+1..j)}}$
   (b) $r_m = j$ and $\sigma' = \sigma\downarrow(r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.

A formula p is said to be:

1. $satisfied$ by an interval σ, denoted by $\sigma \models p$, if $(\sigma, 0, 0, |\sigma|) \models p$.
2. $satisfiable$, if $\sigma \models p$ for some σ.
3. $valid$, denoted by $\models p$, if $\sigma \models p$ for all σ.
4. $equivalent$ to another formula q, denoted by $p \equiv q$, if $\models (p \leftrightarrow q)$.

The abbreviations $true, false, \wedge, \rightarrow and \leftrightarrow$ are defined as usual. In particular, $true \stackrel{\text{def}}{=} P \vee \neg P$ and $false \stackrel{\text{def}}{=} \neg P \wedge P$ for any formula P. Also some derived formulas are shown in Fig.2.

$$
\begin{array}{llll}
empty \stackrel{\text{def}}{=} \neg \bigcirc true & & more \stackrel{\text{def}}{=} \neg empty & \\
halt(p) \stackrel{\text{def}}{=} \Box(empty \leftrightarrow p) & & keep(p) \stackrel{\text{def}}{=} \Box(\neg empty \rightarrow p) & \\
fin(p) \stackrel{\text{def}}{=} \Box(empty \rightarrow p) & & skip \stackrel{\text{def}}{=} \neg empty & \\
x\circ = e \stackrel{\text{def}}{=} \bigcirc x = e & & x := e \stackrel{\text{def}}{=} skip \wedge x\circ = e & \\
len(0) \stackrel{\text{def}}{=} empty & & len(n) \stackrel{\text{def}}{=} \bigcirc len(n-1)(n > 0) &
\end{array}
$$

Fig. 2. Derived formulas

2.2 Propositional Projection Temporal Logic

Let $Prop$ be a countable set of atomic propositions. The formula of PPTL is given by the following grammar:

$$p ::= \pi \mid \bigcirc p \mid \neg p \mid p_1 \vee p_2 \mid (p_1, \ldots, p_m)\, prj\ p \mid p^+$$

where $\pi \in Prop$, $p_1, \ldots, p_m$ are all well-formed PPTL formulars. A formula is called a state formula if it contains no temporal operators.

Following the definition of Kripke structure, we define a state s over $Prop$ to be a mapping from $Prop$ to $B = \{true, false\}$, $s : Prop \rightarrow B$. We will use $s[\pi]$ to denote the valuation of π at state s. Intervals, interpretation and satisfaction relation can be defined in the same way as in the first order case.

2.3 Modeling, Simulation and Verification Language

The language MSVL with frame [8] technique is an executable subset of PTL and used to model, simulate and verify concurrent systems. The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$
\begin{array}{l}
e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0\, op\, e_1 (op ::= + \mid - \mid * \mid / \mid mod) \\
b ::= true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1
\end{array}
$$

where n is an integer and x is a variable. The elementary statements in MSVL are defined as follows:

Assignment:	$x = e$
P-I-Assignment:	$x \Leftarrow e$
Conditional:	if b then p else $q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
While:	while b do $p \stackrel{\text{def}}{=} (b \wedge p)^* \wedge \Box(empty \rightarrow \neg b)$
Conjunction:	$p \wedge q$
Selection:	$p \vee q$
Next:	$\bigcirc p$
Always:	$\Box p$
Termination:	$empty$
Sequential:	$p; q$
Local variable:	$\exists x : p$
State Frame:	$lbf(x)$
Interval Frame:	$frame(x)$
Parallel:	$p \parallel q \stackrel{\text{def}}{=} p \wedge (q; true) \vee q \wedge (p; true)$
Projection:	$(p_1, \ldots, p_m)\ prj\ q$
Await:	$await(b) \stackrel{\text{def}}{=} (frame(x_1) \wedge \cdots \wedge frame(x_h)) \wedge \Box(empty \leftrightarrow b)$ where $x_i \in V_b = \{x \mid x \text{ appears in } b\}$

where x denotes a variable, e stands for an arbitrary arithmetic expression, b a boolean expression, and $p_1, \ldots, p_m$, p and q stand for programs of MSVL. The assignment $x = e$, $x \Leftarrow e$, and $empty$, $lbf(x)$ as well as $frame(x)$ can be regarded as basic statements and the others composite ones.

The assignment $x = e$ means that the value of variable x is equal to the value of expression e. Positive immediate assignment $x \Leftarrow e$ indicates that the value of x is equal to the value of e and the assignment flag, p_x, for variable x is true. Statements $if\ b\ then\ p\ else\ q$ and $while\ b\ do\ p$ are the same as that in the conventional imperative languages. $p \wedge q$ means that p and q are executed concurrently and share all the variables during the mutual execution. $p \vee q$ means p or q are executed. The next statement $\bigcirc p$ means that p holds at the next state while $\Box p$ means that p holds at all the states over the whole interval from now. $empty$ is the termination statement meaning that the current state is the final state of the interval over which the program is executed. The sequence statement $p; q$ means that p is executed from the current state to its termination while q will hold at the final state of p and be executed from that state. The existential quantification $\exists x : p$ intends to hide the variable x within the process p. $lbx(x)$ means the value of x in the current state equals to value of x in the previous state if no assignment to x occurs, while $frame(x)$ indicates that the value of variable x always keeps its old value over an interval if no assignment to x is encountered. Different from the conjunction statement, the parallel statement allows both the processes to specify their own intervals. e.g., $len(2) \| len(3)$ holds but $len(2) \wedge len(3)$ is obviously false. Projection can be thought of as a special parallel computation which is executed on different time scales. The projection $(p_1, \ldots, p_m)\ prj\ q$ means that q is executed in parallel with $p_1, \ldots, p_m$ over an interval obtained by taking the endpoints

of the intervals over which the p_is are executed. In particular, the sequence of p_is and q may terminate at different time points. Finally, $await(b)$ does not change any variable, but waits until the condition b becomes true, at which point it terminates.

Further, the following derived statements are useful in practice.

Multiple Selection:	$OR_{K=1}^{n} \stackrel{\text{def}}{=} p_1 \vee p_2 \vee \ldots \vee p_n$
Conditional:	if b do $p \stackrel{\text{def}}{=}$ if b do p else empty
When:	when b do $p \stackrel{\text{def}}{=} await(b); p$
Guarded Command:	$b_1 \rightarrow p_n \Box \ldots \Box b_n \rightarrow p_n \stackrel{\text{def}}{=} OR_{k=1}^{n}$ (when b_k do p_k)
Repeat:	repeat p until $c \stackrel{\text{def}}{=} p; while \neg c$ do p

2.4 Unified Model Checking Approach

The idea of unified model checking approach [11] is as follows: modeling the system to be verified by an MSVL program p, and specifying the desired property of the system by a PPTL formula ϕ, to check whether or not the system satisfies the property, we need to prove the validation of

$$p \rightarrow \phi$$

If $p \rightarrow \phi$ is valid, the system satisfies the property, otherwise, the system violates the property. Equivalently, we can check the satisfiabbility of

$$\neg(p \rightarrow \phi) \equiv p \wedge \neg\phi$$

If $p \wedge \neg\phi$ is unsatisfiable, $(p \rightarrow \phi)$ is valid, and the system satisfies the property, otherwise, the system fails to satisfy the property. For each $\sigma \models p \wedge \neg\phi$, σ determines a counterexample that the system violates the property. Accordingly, our model checking approach can be translated to the satisfiability of PTL formulas of the form $p \wedge \neg\phi$, where p is an MSVL program and ϕ is a formula in PPTL. Since both model p and property ϕ are formulas in PTL, we call this model checking approach as unified model checking.

To check the satisfiability of PTL formula $p \wedge \neg\phi$, we construct the NFG (Normal Form Graph) of $p \wedge \neg\phi$. As depicted in Fig.3, initially, we create the root node $p \wedge \neg\phi$, then we rewrite p and $\neg\phi$ into their normal forms respectively. By computing the conjunction of normal forms of p and $\neg\phi$, new nodes ϵ and $p_{fj} \wedge \neg\phi_{fs}$, and edges $(p \wedge \neg\phi, p_{ei} \wedge \neg\phi_{ck}, \epsilon)$ from node $p \wedge \neg\phi$ to ϵ, $(p \wedge \neg\phi, p_{cj} \wedge \neg\phi_{cs}, p_{fj} \wedge \neg\phi_{fs})$ from $p \wedge \neg\phi$ to $p_{fj} \wedge \neg\phi_{fs}$ are created. Further, by dealing with each new created nodes $p_{fj} \wedge \neg\phi_{fs}$ using the same methods as the root nodes $p \wedge \neg\phi$ repeatedly, the NFG of $p \wedge \neg\phi$ can be produced. Thus, it is apparent that each node in the NFG of $p \wedge \neg\phi$ is in the form of $p' \wedge \neg\phi'$, where p' and ϕ' are nodes in the NFGs of p and $\neg\phi$ respectively.

In the NFG of formula $q \equiv p \wedge \neg\phi$, a finite path, $\Pi = < q, q_e, q_1, q_{1e}, \ldots, \epsilon >$, is an alternate sequence of nodes and edges from the root to ϵ node, while an infinite path, $\Pi = < q, q_e, q_1, q_{1e} >$, is an infinite alternate sequence of nodes and edges emanating from the root. Similar to the proof in [12], it can be proved that, the paths in the NFG of q precisely characterize models of q. Thus, if there exists a path in the NFG of q, q is satisfiable, otherwise unsatisfiable.

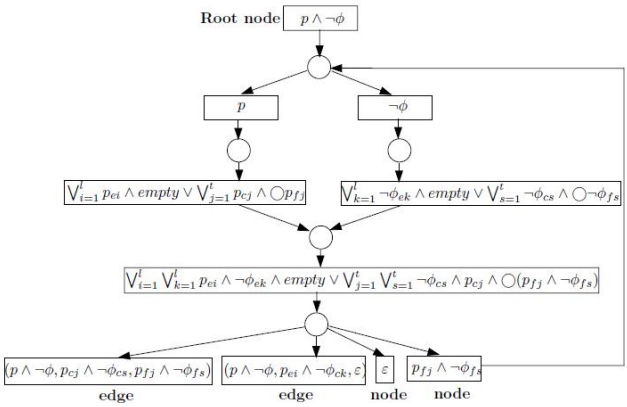

Fig. 3. Constructing NFG of $p \wedge \neg\phi$

3 Translating C Programs into MSVL Programs

In this section, we present the main procedure of transforming a C program into an MSVL program, Fig.4 shows the general process. First of all, the original C program is input into the lexical and syntax analyzers for the analysis. After this process, all useful fragments including identifiers, operators, expressions, and statements in a C program are identified and stored in a specified structure. Finally, we invoke the translation program written in C++ to extract information from the storage structure and to output the translated MSVL program.

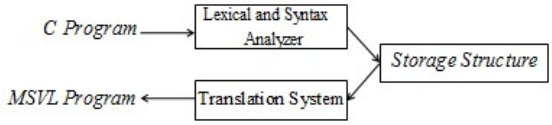

Fig. 4. General Process of the Translation

3.1 Lexical and Syntax Analysis

The first step is to identify the identifiers and statements in the original C program. We accomplish the identification with the help of a common tool named Parser Generator (PG) which integrates Lex and Yacc together. PG is a generator of lexical and syntax analyzers. What we need to do first is to specify the regular expressions for key words (or tokens) and all statements in C program. The regular expressions are defined in two kinds of documents with the extension being .l and .y. Lex source file is a table of regular expressions and corresponding program fragments. It will be translated into a program which reads an input C program stream, copying it to an output stream and partitioning the input stream into tokens which match our given expressions. Yacc source

file specifies the structures of the input, together with code to be invoked as each such structure is recognized, and Yacc turns such a specification into a subroutine. As we can see, based on these two kinds of specific documents, PG will automatically generate an analysis program in C++, which implements the identification of original C programs.

The key step of lexical and syntax analysis is the specification of regular expressions for basic tokens and statements in C programs. A C program normally consists of a list of elementary statements, and for each normative statement, it may contain keywords, basic expressions, sub-statements or a block of statements.

Lex source file specifies the regular expressions of basic constructs in C language, such as variables, characters set, data types, constants, keywords (reserved words in C language), identifiers, arrays and so on. Yacc source file defines regular expressions for elementary C statements, as well as the corresponding handle functions when such statements are matched. Fig.5 shows the basic process of lexical and syntax analysis. At first, source files $lexer.l$ and $parser.y$ are input into PG process. After the analysis, executable programs in C++ are generated. By compiling the programs, we finally obtain an analyzer for C programs.

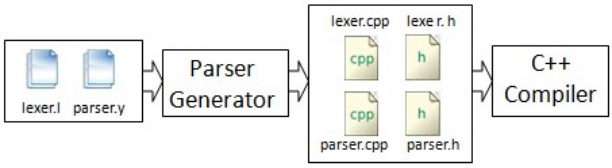

Fig. 5. Process of Lexical and Syntax Analysis

3.2 Storage Structure

To store information, we formalize two classes: one for expressions and statements named AstNode, and the other one for statement blocks named StmtBlock. Member variables of AstNode and StmtBlock are shown in Fig.6 and Fig.7 respectively.

As shown in Fig.6, class AstNode contains four primary member variables: AstNode $*left$, AstNode $*right$, StmtBlock $*block$ and AstNodeType $type$. $left$ and $right$ are used to store basic expressions or sub-statements that may appear in elementary C statements. $type$ is an enumeration variable used to indicate the type of statements or expressions. For instance, *ANT_FOR* represents "for-statement" and *ANT_ADD* stands for plus(+) operation. The remaining member variables such as string str_1 and int int_1 are auxiliaries used to store the name of a variable or the value of an integer variable. Each statement in C programs is related to an instance of class AstNode.

Fig.7 shows the structure of class StmtBlock, it contains two primary member variables: AstNode $*stmt$ and StmtBlock $*block$. Actually, a StmtBlock instance can be regarded as a list of AstNode instances which relates to a statement block.

$$\textbf{AstNode}\begin{cases} AstNode * left \\ AstNode * right \\ StmtBlock * block \\ AstNodeType \quad type \\ string \quad str1 \\ int \quad int1 \end{cases}$$

Fig. 6. Class *AstNode* Structure

$$\textbf{stmtblock}\begin{cases} AstNode * stmt \\ StmtBlock * next \end{cases}$$

Fig. 7. Class *StmtBlock* Struture

Member functions are also needed to manipulate the member variables. Here we mainly introduce function $allco()$ since it is frequently used.

- For class AstNode, $alloc()$ generates an AstNode instance dynamically, meanwhile, the function needs three formal parameters: AstNode $*left$, AstNode $*right$ and AstNodeType $type$.
- For class StmtBlock, $alloc()$ generates a StmtBlock instance dynamically. The function needs only one formal parameter: AstNode $*p$, which points to the head of the AstNode list.

Whenever a regular expression in Yacc source file is matched, corresponding handle functions such as $alloc()$ will be invoked, therefore, statement fragments can be stored in the newborn class instance. Functions dealing with setting and getting member variables are also needed, we omit the details here.

As to the storage of the whole C program, we make use of a container in STL(Standard Template Library) named $vector$, which acts as a dynamic array. A variety of functions are encapsulated in $vector$, so we can call them directly, such as $push\_back()$ and $pop\_back()$. $stmt$ is a container we defined to store all statements in a C program. To store a C fragment in container $stmt$, we only need to call function $stmt.push\_back()$.

3.3 Translation

3.3.1 Translation Algorithm

Algorithm TransForm is formalized to realize the translation. Given a C program fragment S as input, we first check the type of S since different types invoke different methods. The pseudo code of algorithm TranForm are shown in Table.1. In the algorithm, GetStmt is used to transform a C statement into an MSVL statement, while GetExpr will transform a basic expression into an MSVL expression. GetBlock aims at the transforming of a statement block. The general process of the algorithm is given below:

- Input required C fragments into the translation procedure
- Specific methods will be invoked
- These methods translate C fragments and output MSVL fragments

Table 1. Algorithm for translating C fragments to MSVL fragments

Function TransForm(S)
/*precondition: S is an elementary fragment of C program */
/*postcondition: Transform(S) computes an equivalent MSVL fragment*/
begin function
case
S is a statement: **return** GetStmt(S);
S is an expression: **return** GetExpr(S);
S is a statement block: **return** GetBlock(S);
end case
end function

In algorithm GetStmt, basic statements such as integer declaration, if statement, while statement, for statement, printf and scanf need to be considered. Intuitively, ";" represents an empty statement, "exp;" stands for an expression statement and "x=exp;" is an assignment statement. However, in order to improve the efficiency of GetStmt, some auxiliary methods such as GetIfStmt and GetDeclaration are also required. The algorithm is straightforward which includes all cases and uses the corresponding rules to transform related fragments as shown in Table.5 and Table.6.

Table 2. Algorithm for translating a C statement to an MSVL statement

Function GetStmt(S)
/*precondition: S is an elementary C statement */
/*postcondition: GetStmt(S) computes an equivalent MSVL statement*/
/* exp is a standard expression, $block$ is a statement block, s represents a complete statement, x is a variable, $parameter$ stands for a string */
begin function
case
S is ; **return** ;
S is exp; **return** exp;
S is x=exp; **return** x:=$exp\ and\ skip$;
S is $if\text{-}statement$: **return** GetIfStmt(S);
S is $while(exp)\{block\}$: **return** $while(exp)\{block\}$;
S is $int\text{-}declaration\text{-}statement$: **return** GetDeclaration(S);
S is $for(s\ exp_1;exp_2)\{block\}$: **return** $s\ while(exp_1)\{block;exp_2;\}$;
S is $printf(\text{“}parameter\text{”},exp)$: **return** $output(exp)$;
S is $scanf(\text{“}parameter\text{”},exp)$: **return** $input(exp)$;
end case
end function

Table 3. Algorithm for translating C expressions to MSVL expressions

Function GetExpr(S)
/*precondition: S is an elementary C expression */
/*postcondition: GetExpr(S) computes an equivalent MSVL expression*/
/*x and y are standard expressions*/
/*e represents a constant, a variable or an identifier */

begin function
case
S is e: **return** e
S is $x\circledast$ ($\circledast$ =[++|- -]): **return** x:=x+1 $and\ skip \mid x$:=x-1 $and\ skip$
S is x=y: **return** x:=y $and\ skip$
S is x==y: **return** x=y
S is x[+,-,*,/,%,!=]y: **return** x[+,-,*,/,%,!=]y
S is x*=y (*=[<|>]): **return** x=y or x*y
S is x[+|-|*|/|%]=y: **return** x:=x[+|-|*|/|%]y $and\ skip$
S is x&&y: **return** x and y
S is $x||y$: **return** x or y
S is x,y: **return** x,y
S is (x): **return** (x)
end case
end function

As to algorithm GetExpr given in Table.3, elementary C expressions are considered, such as identifier, constant, arithmetic expression, logical expression, relational expression, bracket expression and so on. Table.4 displays the transformation rules of algorithm GetBlock. Since a statement block is a list of statements linked by pointers, to transform a state block we can repeatedly call GetStmt(S) as long as S is not $null$. $next(S)$ in algorithm GetBlock means that S points to the next statement.

Table 4. Algorithm for translating a statement block

Function GetBlock(S)
/*precondition: S is an elementary C statement block */
/*postcondition: GetBlock(S) computes an equivalent MSVL block*/

begin function

repeat{GetStmt(S); S=$next(S)$;} **until** (S=null;)

end function

Table 5. Algorithm for translating $if\text{-}statement$

Function getIfStmt(S) /*precondition: S is an elementary if-statement in C language */ /*postcondition: getIfStmt(S) computes an equivalent MSVL if-statement*/ /*exp is an standard expression and $block$ is a statement block*/
begin function **case** S is $if(exp)s$: return $if(exp)then\ \{s\}$ S is $if(exp)\{block\}$: return $if(exp)then\{block\}$ S is $if(exp)\{block_1\}else\{block_2\}$: return $if(exp)then\{block_1\}$ $else\{block_2\}$ S is $if(exp_1)\{block_1\}else\ if(exp_2)\{block_2\}$: return $if(exp_1)then$ $\{block_1\}else\ if(exp_2)\{block_2\}$ **end case** **end function**

Table 6. Algorithm for Translating $int\text{-}declaration\text{-}statement$

Function getDeclaration(S) /*precondition: S is an elementary int-declaration-statement in C language */ /*postcondition: getDeclaration(S) computes an equivalent MSVL int-declaration-statement*/ /*var_list represents a list of variables connected by "," */ /*exp_list represents a list of expression connected by ","*/ /*e_i is one of the expressions that appear in exp_list*/
begin function **case** S is $int\ x$;: return $int\ x$; S is $int\ var\_list$;: return $int\ var\_list$; S is $int\ x$=exp;: return $int\ x$; x:=$exp\ and\ skip$; S is $int\ exp\_list$;: return getDeclaration(e_i); ($0<i<=len(exp\_list)$) **end case** **end function**

3.3.2 Implementation

We now focus on the implementations of algorithm TransForm, which includes $GetStmt()$, $GetExpr()$ and $GetBlock()$. The algorithm has two kinds of input parameters: $AstNode$, $StmtBlock$, and different parameters invoke different methods. Since $GetStmt()$ and $GetExpr()$ have much in common, we omit the details for $GetExpr()$ here. Brief descriptions of $GetStmt()$ and $GetBlock()$ are shown in Fig.8 and Fig.9. As we can see, method $Switch$ is the core algorithm for implementations. It generates

Method: $GetStmt()$
Parameter Type: $AstNode$
Function: Realize the translation and output of elementary statements
Process:

- $GetType()$ is called to get $type$, which specifies the type of a statement.
- Call $Switch(type)$ to realize the transformation and output

Fig. 8. GetStmt()

Method: $GetBlock()$
Parameter Type: $StmtBlcok$
Function: Realize the translation and output of a statement block
Process:

- A repeated calling of $GetStmt()$

Fig. 9. GetBlock()

different results according to parameter $type$. $type$ is more like a statement label which informs $Switch$ of which transformation rule should be effective .

So far, the transformation from basic C programs to MSVL programs is almost done. However, some problems are worth paying attention in order to make sure that the final output fits the MSVL interpreter well. What we need to point out here is an obvious difference between C and MSVL: in C programs, the last statement in a block must ends with a semicolon(;), however, in MSVL the semicolon has been saved. To take both situations into consideration, a method named $GetStmt\_last()$ is designed to cope with this problem. $GetStmt\_last()$ is totally as the same as $GetStmt()$ except that the former gets rid of the semicolon at the end of a statement.

4 A Case Study

In this section, we will give a basic but typical C program which consists of a number of elementary C statements. Then, by employing our translation algorithm, we can get an equivalent MSVL program.

4.1 Greatest Common Divisor and Lowest Common Multiple

There are many ways to find the Greatest Common Divisor (GCD) and the Lowest Common Multiple (LCM) of two positive integers. Here we use Euclidean Algorithm (Euclid's Algorithm) to find GCD and a particular formula to calculate LCM of two positive integers.

4.1.1 Euclidean Algorithm

This algorithm finds GCD by performing repeated division starting from the two numbers we want to find out the GCD until we get a remainder of 0. Below are the steps to compute GCD of positive integers, 12 and 8, by using Euclid's algorithm.

- Divide the larger number by the small one. In this case we divide 12 by 8 to get a quotient of 1 and remainder of 4.
- Next we divide the smaller number (i.e. 8) by the remainder from the last division (i.e. 4). So 8 is divided by 4, and we get a quotient of 2 and remainder of 0.
- Since we already get a remainder of 0, the last number that we used to divide is the GCD, i.e 4.

Implementations of the algorithm may be expressed in pseudo code. For example, the division-based version may be programmed as below:

```
function gcd(a, b)
while b != 0
t := b
b := a mod b
a := t
return a
```

4.1.2 A Formula to Find LCM

We can use the following formula to calculate LCM of positive integer a and b if we already know $GCD(a, b)$.

$$\text{LCM(a,b)} = \frac{\text{a} \times \text{b}}{\text{GCD(a,b)}}$$

Fig. 10. Formula for Calculating LCM

4.2 Translation from C to MSVL

The implementation code in Fig.11 shows how to find GCD and LCM of two positive integers: a and b. It is based on the algorithm we mentioned in section 4.1. Note that the code contains elementary C statements, such as variable declaration, assignment statement, arithmetic expression, if statement, while statement, printf statement and scanf statement. After employing the transformation algorithm to the original C program, an equivalent MSVL program is generated as shown in Fig.12.

To test the correctness of the MSVL program, as illustrated in Fig.13, we use two positive integers: 256 and 60. According to the execution result, GCD(256,60) equals to 4 and LCM(256,60) equals to 3840, which is apparently correct.

```
#include<stdio.h>
int main()
{
  int num1,num2,a,b,t;
  printf("please input two numbers:");
  scanf("%d %d",&a,&b);
  if(a<b){
       t=a;
       a=b;
       b=t;}
  num1=a;num2=b;
  while(num2!=0){
       t=num2;
       num2=num1%num2;
       num1=t;}
  printf("GCD:%d\n",num1);
  printf("LCM:%d\n",(a*b)/num1);
  return 0;
}
```

Fig. 11. C Program

```
frame(num1,num2,a,b,t) and skip;
int num1,num2,a,b,t;
output("please input two numbers:") and skip;
input(a) and input(b)and skip ;
if ( a<b ) then{
t:= a and skip;
a:= b and skip;
b:= t and skip
} and skip;
num1:= a and skip;
num2:= b and skip;
while(num2!=0){
t:= num2 and skip;
num2:= num1 % num2 and skip;
num1:= t and skip
};
output ("GCD:",num1) and skip ;
output ("LCM:", (a*b)/num1) and skip
```

Fig. 12. MSVL Program

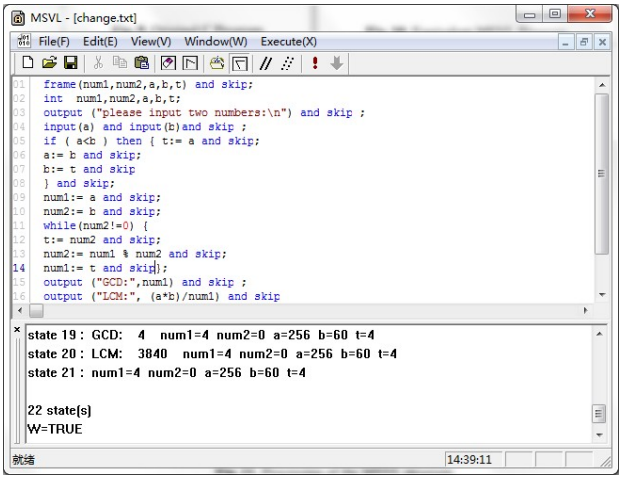

Fig. 13. Executing of the Final MSVL program

4.3 Verification with MSVL

After the translation, we can apply the existing unified model checking approach to the MSVL program. Since the C program and the MSVL program are equivalent, the validity of the MSVL program also indicates the validity of the original C program. As to our example, two positive integers are 256 and 60. It is easy to find out that the property "final GCD either less than 60 or equals to 60" should always hold. By employing propositions p and q to denote $gcd{<}small$ and $gcd{=}small$ respectively,

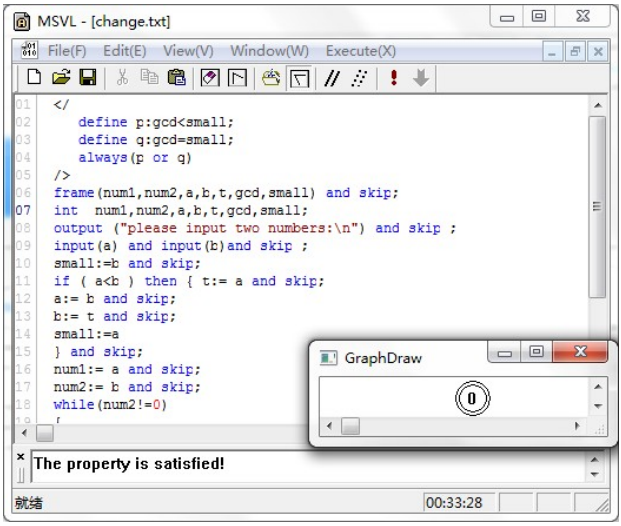

Fig. 14. Verification Result

this property can be specified by $\Box(p \vee q)$ in PPTL. Under the verification mode of MSVL, we add the following code

```
</
  define p:gcd<small;
  define q:gcd=small;
  always(p or q)
/>
```

to the beginning of the MSVL program. In this property, gcd represents the final GCD of the two positive integers, $small$ is the smaller integer of the two positive integers. After running the program, an empty NFG with no edge is produced as shown in Fig.14. Hence,the formula is unsatisfiable, and the system satisfies the property.

5 Conclusion

In this paper, we present a translator from C programs to MSVL programs. Therefore, by transforming a C program into an equivalent MSVL program, we can employ the existing techniques to verify the correctness of C programs. This enables us to translate the problem of checking whether or not the C program satisfies the property to the problem of checking the satisfiability of MSVL programs.

However, we just realize the transformation from basic C programs to MSVL programs at the moment. For some complex C programs, further investigations are still needed. Currently, the translator is merely a prototype, and lots of efforts are needed to improve it. In addition, to examine our method, several big case studies are required in the near future.

References

1. Ostroff, J.S.: Verification of safety critical systems using TTM/RTTL. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 573–602. Springer, Heidelberg (1992)
2. Yang, M., Wang, Z., Pu, G., Qin, S., Gu, B., He, J.: The Stochastic Semantics and Verification for Periodic Control Systems. Science China: Information Sciences 55(12), 1–19 (2012)
3. Qin, S., Luo, C., Chin, W.-N., He, G.: Automatically Refining Partial Specifications for Program Verification. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 369–385. Springer, Heidelberg (2011)
4. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)
5. Ghezzi, C., Mandrioli, D., Morzenti, A.: Specifying real-time properties with metric temporal logic. J. Syst. Softw. 12(2), 107–123 (1990)
6. Jahanian, F., Mok, A.K.: Safety analysis of timing properties in real-time systems. IEEE Trans. Softw. Eng. SE-12(9), 890–904 (1986)
7. Duan, Z.: An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD Thesis, University of Newcastle upon Tyne (1996)
8. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2006)
9. Alur, R., Henzinger, T.A.: A really temporal logic. In: Proceedings of the 30th IEEE Conference on Foundations of Computer Science. IEEE Computer Society Press, Los Alamitos (1989)
10. Melliar-Smith, P.M.: Extending interval logic to real time systems. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 224–242. Springer, Heidelberg (1989)
11. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008)
12. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Informatica 45(1), 43–78 (2008)

An Application of SOFL for Rapid Prototyping

Fumiko Nagoya and Tetsuo Kitagawa

Graduate School of International Management, Aoyama Gakuin University, Japan
fnagoya@gsim.aoyama.ac.jp, tkitagawa@aoyamagakuin.jp

Abstract. A prototype is a model of a product or information system to show the capabilities. It is common practice to build prototypes in agile software development to help end-users and developers understand the requirements elicitation and validation for a system. Though the prototypes can serve with behaviors and structures, some requirements such as safety-critical functions are difficult to prototype. Formal methods are used to reveal ambiguity, inconsistency and incompleteness for development computer systems with commonly used formal specification languages. Formal methods are mathematically-based techniques. Nevertheless few specifiers have used complete mathematics from the beginning. Most of specifiers have discussed with end-users about implications of user requirements, created an initial specification, and revised as a result of the user's feedbacks. This paper presents an experimental project adapting rapid prototyping into a formal engineering method. SOFL, the formal engineering method, serves as a bridge between formal methods and rapid prototyping. It has great potential to improve development costs and product quality.

1 Introduction

Prototyping has been widely known after the publication of Brooks' The Mythical Man-Month. Prototyping is viewed as the process which is a well-defined phase in software development [1]. It allows that "final implementation is developed that meets the project specifications" [2]. Additionally, it makes it possible to increase effective communication [3], between end-users and developers. In particular, software prototyping has substantial advantages over hardware, to decrease development time and costly mistakes [4], [5], hence the name rapid prototyping [6]. Since, a number of different definitions, processes, and classifications for rapid prototyping have been provided by researchers and developers. Following Floyd [1], all of them are classified based on goals of prototyping: *exploratory, experimental, or evolutionary prototyping.*

Balzer et al., have visualized a future of automation-based software paradigm [7] as integrating prototypes into formal specifications. In this paradigm, the specifications are created and maintained by end-users. Subsequently, the revised specifications become prototypes of desired system, and the prototypes are validated and implemented by machine aided. Such implementation process makes it possible to be fast, reliable, and inexpensive software development. Unfortunately, regardless of many researcher's contributions and efforts, the automation-based

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 104–115, 2013.

software paradigm is not feasible. Because, the wide gap between formal methods and actual developments in industry has not yet resolved.

This paper addresses the gap, and proposes a model for integrating rapid prototyping and formal engineering methods. Formal engineering methods are designed to serve as a bridge between formal methods and developments of applications. Structured Object-Oriented Formal Language (SOFL) is a comprehensible language for requirements and design specifications [8]. It provides a practical notation using Data Flow Diagram [9], Petri nets [10], and VDM-SL [11]. Also, SOFL is a method for combination of structured methods and object-oriented methods to construct specifications. Structured methods [12], [9] induce functional decomposition of the top level module into a hierarchy of low level modules by top-down approach. Object-oriented methods offer an approach to discover the classes and objects that form the vocabulary of the problem domain, identifying the semantics and the relationships among them, and specifying the interface and the implementation of these classes and objects [13]. SOFL facilitates the functional decomposition and translation into object-oriented programming such as C++ and Java.

Our ongoing project is an application software development. The application offers a working model for forecasting financial statements and calculating business valuations by two materials: disclosed documents in the Financial Services Agency of the Japanese Government, and users' expectations. In this project, main reason for using a rapid prototyping is driving implicit knowledge of a domain expert.

The remainder of this paper is organized as follows. Section 2 briefly introduces rapid prototyping, our proposed development model, and key factors for rapid prototyping. Section 3 describes the background and motivation, and elaborates how to use SOFL for this project. Section 4 explains our experience, although this project has not been completed yet. Lastly, in section 5 we give conclusions and point out future research.

2 Rapid Prototyping

Rapid prototyping is the activity which occurs early in the software development life cycle [14]. Software development life cycle is a process to create software systems. Various software development models exist, for example, *waterfall model*, *spiral model*, *prototyping model*, etc.

Royce [15] described the *waterfall model* that software development life cycle consists of several steps from system requirements to operation of the products, and each steps should complete before their successors begin. The *spiral model* of software development life cycle was defined by Boehm [16] as iterative development model. It has characteristics of both the *waterfall model* and *prototyping model*. In this section, we describe *prototyping model* and our proposed *formal engineering prototyping model* at first. These *prototyping model* and *formal engineering prototyping model* have the following features: 'user participation' and 'iterative procedure'. We mention the features of successful rapid prototyping, continuously.

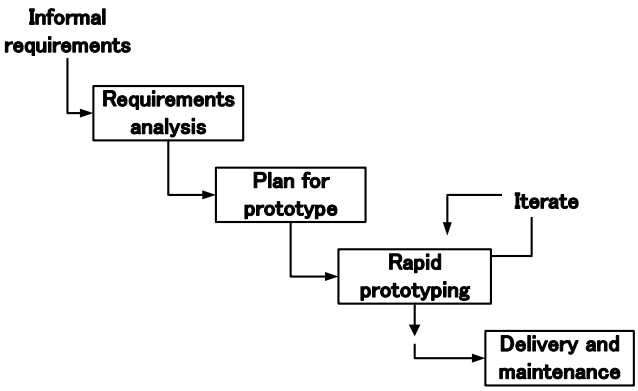

Fig. 1. Prototype Model

2.1 Formal Engineering Prototyping Model

The *prototyping model* is represented in Figure 1. The typical *prototyping model* comprises four phases: *requirements analysis*, *plan for prototype*, *rapid prototyping*, and *delivery and maintenance*. This model has flexibility to unforeseen changes compared with *waterfall model* and *spiral model*. In the *prototyping model*, it is possible to check whether the prototype meets user requirements by demonstration. However, it is impossible to develop safety-critical system by lack of verification, and difficult to maintain for long-term as there is no specification.

Figure 2 shows our suggested *formal engineering prototyping model*. It is a combination of *prototyping model* and continuous phases: *formal specification*, *coding*, *testing*, and *delivery and maintenance*. In this model, a rapid prototype serves as an informal specification. In other words, the rapid prototype guides to write a formal specification. The model may receive the following benefits of formal methods. A formal method forces to be precise [17], reveal ambiguities, inconsistencies, and incompleteness [18], and supports automatic or machine-assisted analysis for verification and validation.

2.2 User Participation

The user of 'user participation' should be included not only a sample of the actual future users, but also domain experts. Because, users may not have a clear idea what they want the new system to do. Additionally users may feel that it is hard to describe their knowledge of the system domain. On the other hand, domain experts have knowledge and experiences of the system domain, and quite familiar with the user's requirements, expectations, and behaviors. It is so important that a domain expert is taking part in system development team. However, Loucopoulos and Karakostas have pointed out that it is difficult for developers to elicit the knowledge from its source, especially when the source is a human 'expert' [19]. Developers are faced with a major difficulty in obtaining

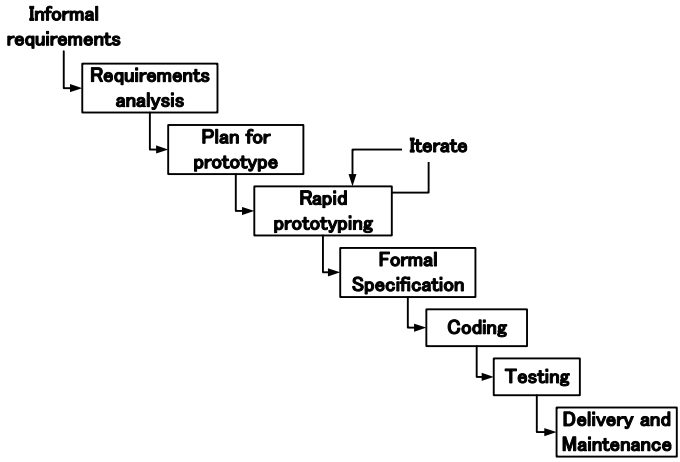

Fig. 2. Formal engineering prototyping model

good understandings of the program domain. It is caused by communication problems that domain experts and system engineers use different languages. Iterative procedure of rapid prototyping contributes to solve the communication problems.

2.3 Iterative Procedure

The 'iterative procedure' means the repeating cycle that a rapid prototype is reviewed by users whether or not the prototype matches their requirements, and improved based on user's feedback. Naumann and Jenkinshave described prototyping is a four-step procedure between user and developer [20]. Figure 3 illustrates the four-step procedure which is repeated until the user accepts next version.

1. Identify the User's Basic Information Requirements
   A developer elicits user's basic needs including data requirements, report formats, user interfaces.
2. Develop a Working Prototype
   The developer quickly creates a working model for the system. The initial prototype includes user interfaces and/or database.
3. Implement and Use the Prototype System
   The developer demonstrates the prototype to the user. The user try to use the prototype and takes notes for their feedback they would like made.
4. Revise and Enhance the Prototype System
   The developer and user discuss desired changes based on the user's feedback. Changes and enhancements for new version are determined. The developer creates next version and goes back to *Implement and Use the Prototype System* step until the user is satisfied with the revised prototype.

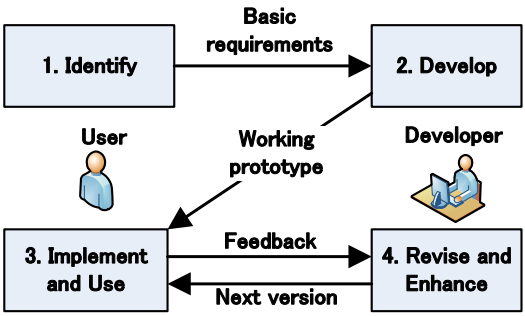

Fig. 3. Four step procedure

This iterative procedure might be support by an appropriate rapid prototyping tool. Such tools design screens, create a simple user interface, display animations, and/or add annotations. A great number of rapid prototyping tools are available, but no one is perfect. The following sections will explains an experimental project and how to perform in each phase of *formal engineering prototyping model* step by step.

3 Experimental Project

This section introduces our ongoing application development project. This application will offer building a financial simulation model for forecasting financial statements and calculating business valuations based on the financial reports and user's expectations. This project has been conducted by a small team consisting of first author as a developer and second author as a domain expert. The developer is used to using the SOFL in order to software development, and the domain expert has more than three decades of experience both sell side and buy side investment analyst in marketable security. First, we describe our background and motivation for this project. Then, we explain our detail project about *requirements analysis*, *plan for prototype*, *rapid prototyping, and formal specification* continuously.

3.1 Background and Motivation

The financial reports have been available to see online in the form of HTML from Electronic Disclosure for Investors' NETwork (EDINET) system operated by the Financial Services Agency of the Japanese Government. The EDINET system is similar to the Electronic Data-Gathering, Analysis, and Retrieval system (EDGAR) system used by the U.S. Securities and Exchange Commission. Some disclosure documents are permitted to be filed electronically by issuers of listed or others who are required by laws. These documents are disclosed online to increase the efficiency and fairness of the securities market.

In past days, general individual investors and other stakeholders who tried to conduct business analysis were difficult to get digital data except professional security analysts using online database such as Bloomberg terminal service. In April 2008, the Japanese Government published a rule for the mandatory use of XBRL(:eXtensible Business Reporting Language) in reporting financial information on EDINET. Past half decade of XBRL files for Japanese companies listed on stock exchanges are available download from EDINET. After online disclosure of financial reports, some of software tools have supported to business analysis, but these functions are unsatisfactory and still expensive for personal use. Then, we have decided to develop a system for forecasting financial statement for our target users who are individual investors and MBA students learning business analysis.

3.2 Requirements Analysis

Figure 4 sketches a whole architecture of the application based on informal requirements. The software uses XBRL file as input data. XBRL is a XML-based languages, and it is an open technology standard for reporting and analyzing business and financial information. A XBRL file contains four primary financial statements (F/S) : the income statement (I/S), the balance sheet (B/S), the statement of cash flows (C/S) and statement of changes in net assets, but it does not include note and annexed detailed statement in Japan. These statements over three or five-years period are required as raw materials of an analysis in order to make sure consistency of accounting policies [21].

This application will provide the following features.

1. Read and parse XBRL files.
2. Save data of historical F/S by accounting period.
3. Display historical B/S, I/S, C/S, and calculated financial indicators.
4. Provide a panel for inputting user's assumptions
5. Calculate future free cash flows based on user's assumptions.
6. Create forecasting B/S, I/S, C/S, and display them.
7. Calculate business valuations and display it.

The financial indicators represent ratio analysis that relates I/S, B/S, C/S to one another. The indicators provide a useful clue as to evaluate a company's current position and judge reasonable projections. The user's assumptions are key drivers to this model. The plane for inputting user's assumptions supplies a questionnaire to assess percentage changes in growth of revenues, profits, and cash flows based on user's projection including macro-economic forecast, capital markets analysis, industry outlook and company business analysis [22]. The business values are computed on the basis of present values of all its future cash flows and risks.

3.3 Plan for Prototype

In this phase, we settled a rough schedule for prototyping, prototyping tool, and scope of the prototyping. The development time for this prototype was predicted

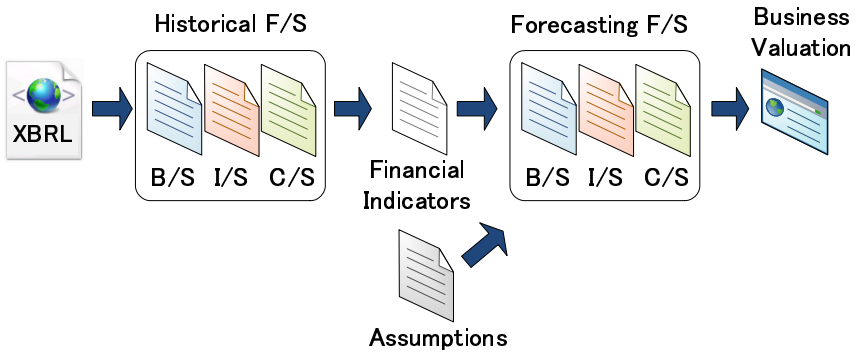

Fig. 4. Software architecture

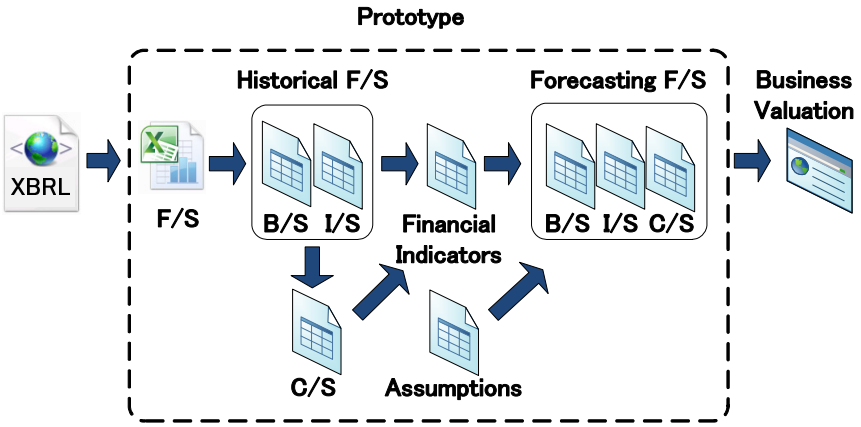

Fig. 5. Scope of prototyping

two or three weeks. To prototype successfully, we chose Microsoft Excel 2010 (Excel) as rapid prototyping tool. The main reason we used Excel is familiar with the domain expert.

The scope of the prototyping is illustrated by Figure 5. There are three differences of functions between the prototype and the user requirements. First, the prototype does not convert from XBRL file to Excel file, and uses Excel data as input. The fully converted file in Excel is available from web services or existing tools. Secondly, we need to generate C/S from historical B/S and I/S for measuring future cash flows. As we mentioned above, the business value depend on the future cash flows. Unfortunately, the regular F/S does not provide for the future cash flows automatically. Third, the prototype provides only basic functions to calculate future cash flows in forecasting years. It means that this application will not implement a complicated business valuation method except net present value method.

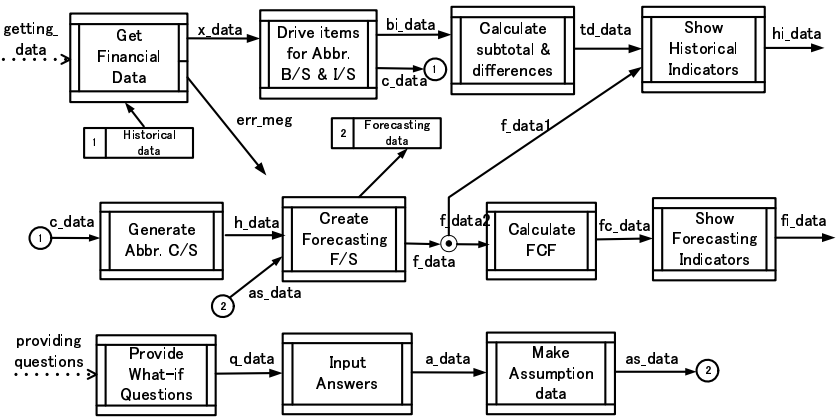

Fig. 6. A top level CDFD

3.4 Rapid Prototyping

The initial prototype was provided by the domain expert. The prototype was a simple spreadsheet and all amounts were based on temporary data for a fictitious company. But it made certain that the developer understands necessary financial items in analysis point of view, question items in assumptions, expressions for calculating financial indicators.

The developer changed the initial prototype through 'iterative procedure' as following:

- The prototype uses real data converted into Excel file from XBRL file.
- The future cash flows are derived from all items of B/S, Net Incomes, and Depreciation Expense in I/S, and Cash Dividend in C/S.
- The prototype displays abbreviated financial statements for a comprehensive look-and-feel.

The domain expert confirmed the revised prototype and returned his feed-backs. The developer learned desired functions, user interface, and domain knowledge about investments in marketable security. The prototype was updated every week for one month through iterative procedure.

3.5 Formal Specification

After the prototype was completed, the formal specification have been developed using SOFL [8] specification language. A SOFL specification consists of a collection of related modules in a hierarchical fashion. Modules are mainly used to express a functional abstraction. Each module encapsulates of processes (like methods in Java), data flows, and data stores (like files or database). Also a module has a condition data flow diagram (CDFD) to represent its functional behavior. The developer began to construct the SOFL specification based on the prototype as below.

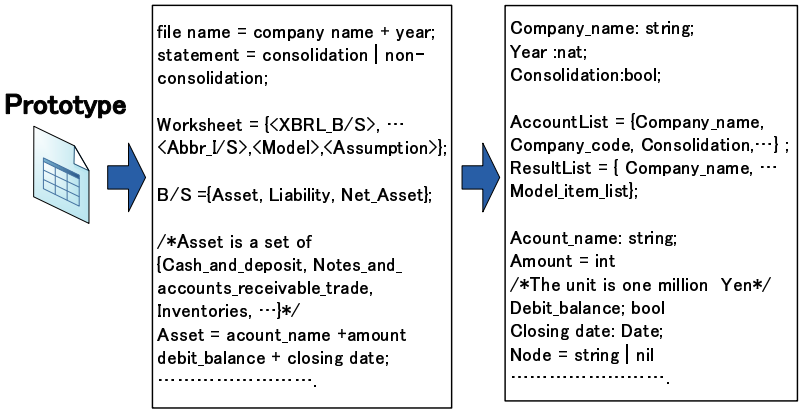

Fig. 7. Stepwise data refinement

1. A top level CDFD was drawn as the architecture of the prototype.
2. All attributes of objects in prototype were defined by a stepwise method.
3. The high level module decomposed into low level operations as processes.

A CDFD is a graphical notation associated with the architecture of specification. The top level CDFD which is derived by the functional behaviors of our prototype is depicted in Figure 6. Each box in this diagram denotes a process, each arrowed line denotes a data flow, and the box with number represent data store. The solid directed denotes an active data flow, while the dotted line denotes a control data flow. Numbered circle represents connecting node, a circle with a dot inside denotes a broadcasting node. The top level CDFD suggests primarily constructive information statically. It assists in writing a formal textural notations based on the predicate logic. A prototype is suited for dynamic representation, but not be enough for showing constructional design.

Figure 7 shows the stepwise data refinement which can be used to facilitate the construction of formal specifications. First, items of raw and columns, worksheet names and file names in the prototype are found as related objects. For example, the item of cash in B/S worksheet have several attributes like name, account balance, closing date, kind of currency, etc. After finding related objects, all attributes consisted of the related objects in our prototype are defined clearly as middle panel. The attributes are finally utilized for type definitions of input, output, and external variables in operation specifications. Therefore, each attribute presents how declared types and constraints would be used to write the specifications as right panel.

Figure 8 shows that the top level module breaks up into low level operations as processes. The top-left graph represents a part of the top level CDFD illustrated in Figure 6. The top level CDFD corresponds to the module '**Forecasting system'** written in the top-right text box. To reduce of the complexity of a one level CDFD, SOFL supports the construction of hierarchical CDFDs and the associated modules by process decomposition [8]. The process '**Generates**

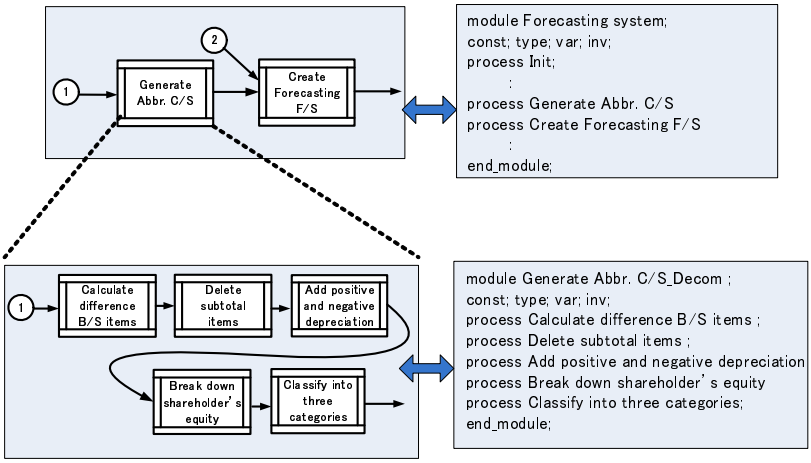

Fig. 8. Process decomposition

Abbr. C/S' is decomposed into the CDFD pictured by the bottom-left graph in Figure 8, and its associated the module which named '**Generates Abbr. C/S_Decom'** described in the bottom-right text box.

4 Experience

Our project has finished only requirements analysis, plan for prototype, rapid prototyping, and formal specification phase. However, we have obtained the experience as following.

- A prototyping tool known how to use by both a domain expert and developers ensures a smoother communication. If the domain expert and developers work closely together without having common understanding of a prototyping tool, it is impossible to serve for efficient elicitation of user requirements and early detection of over and short functions against the user's intentions.
- A scope of prototype leads to be completed on schedule. A weakness of rapid prototyping is that desires to produce better software products are no limit, however the development cost is depend on times. Therefore we should decide what point to stop for developing the prototype, in advance.
- SOFL powerfully supports to transform from the prototype into the formal specification. It facilitates to clarify the whole structure of software by graphical notations, to define all attributes of objects in prototype by stepwise data refinement, and to decompose a high level operation into low level operations.

We have also discovered the difficulty in the formal engineering prototyping modeling.

– It is difficult to validate that the contents defined by the formal specification match the corresponding prototype. A validation methodology is needed to ensure the consistency and completeness.

5 Conclusions and Future Research

We have presented a formal engineering prototyping model using SOFL to serve as a bridge between formal methods and rapid prototyping. The model consists of requirements analysis, plan for prototype, rapid prototyping, formal specification, coding, testing, and delivery and maintenance. In this paper, we have explained concrete activities for our ongoing application development project, and included illustrations of each phase for the requirements analysis, plan for prototype, rapid prototyping, and formal specification. SOFL have assisted to transform from the rapid prototyping into a formal specification.

There is no guarantee that the model fits in any software project. But our experience suggests that our model has great potential to improve development costs and product quality with the benefit of rapid prototyping and formal methods. We assume there are three keys to realize the automation-based software paradigm : prototyping tool, scope of prototype, and formal specification.

To investigate the effectiveness of the model adapting rapid prototyping into SOFL, we need to conduct case studies for verification in our ongoing project. Additionally, we have faced the difficulty to validate whether the formal specification matches the corresponding prototype. Our future research is to figure out a solution for these verification and validation problems between rapid prototyping and formal specifications.

References

1. Floyd, C.: A systematic look at prototyping. In: Approaches to Prototyping, pp. 1–17 (1984)
2. Basili, V.R., Turner, A.J.: Iterative enhancement: A practical technique for software development. IEEE Trans. Software Eng. 1, 390–396 (1975)
3. Alavi, M.: An assessment of the prototyping approach to information systems development. Commun. ACM 27, 556–563 (1984)
4. Boehm, B.W., Gray, T.E., Seewaldt, T.: Prototyping versus specifying: A multiproject experiment. IEEE Trans. Softw. Eng. 10, 290–302 (1984)
5. Gordon, V.S., Bieman, J.M.: Rapid prototyping: Lessons learned. IEEE Softw. 12, 85–95 (1995)
6. Gomaa, H.: The impact of rapid prototyping on specifying user requirements. SIGSOFT Softw. Eng. Notes 8, 17–27 (1983)
7. Balzer, R., Cheatham Jr., T.E., Green, C.: Software technology in the 1990's: Using a new paradigm. Computer 16, 39–45 (1983)
8. Liu, S.: Formal Engineering for Industrial Software Development. Springer (2004)
9. DeMarco, T.: Structured Analysis and System Specification. Prentice Hall PTR, Upper Saddle River (1979)
10. Reisig, W.: Petri nets: An introduction. Springer-Verlag New York, Inc., New York (1985)

11. Jones, C.B.: Systematic software development using VDM. Prentice Hall International (UK) Ltd. (1986)
12. Yourdon, E.: Modern structured analysis. Yourdon Press (1989)
13. Booch, G.: Object oriented design with applications. Benjamin-Cummings Publishing Co., Inc., Redwood City (1991)
14. Gordon, V.S., Bieman, J.M.: Reported effects of rapid prototyping on industrial software quality. Software Quality Journal 2, 93–108 (1993)
15. Royce, W.W.: Managing the development of large software systems: Concepts and techniques. In: Proceedings of the 9th International Conference on Software Engineering, ICSE 1987, pp. 328–338. IEEE Computer Society Press, Los Alamitos (1987)
16. Boehm, B.W.: A spiral model of software development and enhancement. Computer 21, 61–72 (1988)
17. Morgan, C.: Programming from specifications. Prentice-Hall, Inc. (1990)
18. Clarke, E.M., Wing, J.M.: Formal methods: State of the art and future directions. ACM Comput. Surv. 28, 626–643 (1996)
19. Loucopoulos, P., Karakostas, V.: System Requirements Engineering. McGraw-Hill, Inc., New York (1995)
20. Naumann, J.D., Jenkins, A.M.: Prototyping: The new paradigm for systems development. MIS Q. 6, 29–44 (1982)
21. Sengupta, C.: Financial Analysis and Modeling Using Excel and VBA (Wiley Finance). Wiley (2009)
22. Hooke, J.C.: Security Analysis and Business Valuation on Wall Street + Companion Web Site: A Comprehensive Guide to Today's Valuation Methods (Wiley Finance). Wiley (2010)

Applying SOFL to a Generic Insulin Pump Software Design

Chung-Ling Lin, Wuwei Shen, and Dionysios Kountanis

Department of Computer Science,
Western Michigan University

Abstract. Software embedded into medical devices demands a higher standard on its safety, as compared to most commercial software. One of the most important reasons is that the safety issue should be thoroughly investigated. In the United States, Food and Drug Administration (FDA) is entitled to scrutinize medical devices to ensure they are safe to the public before they enter the market. However, the review of medical device software has been quite challenging because not only the design of medical device software is complicated and error-prone but also the validation of the software system against regulatory requirements is notoriously difficult. Thus, some methodologies based on formal methods have been proposed to alleviate the pain faced by both software developers and regulators such as FDA staff. In this paper, we study how to use the Structured-Object-Based-Formal Language, which is called SOFL to develop a software system controlling an insulin pump, called the *Generic Insulin Infusion Pump* (GIIP). This case study facilitates the understanding of how SOFL can be applied to software systems related to medical devices in terms of the design and review aspects.

1 Introduction

One of the most challenging issues facing the software engineering community is how to develop a software system that, software engineers can guarantee, satisfies the specified requirements. To support any type of guarantee, there is an implicit need to establish sufficient *evidence* that the system will perform dependably, as intended. The quality of this evidence can be a key factor in regulator and third party assessments of dependability claims. This need is particularly important for safety and security critical products such as medical devices.

To complicate matters further, as systems evolve to meet new demands, it is essential to be able to establish evidence that any changes made to the system do not affect the integrity of existing dependability properties and introduce new errors in the process.

Traditional software development methodologies emphasize process oriented practices as a means of assuring design artifacts are complete and consistent. This practice often leads to ambiguities within and between requirements and subsequent specifications. The software development landscape is littered with failures rooted in this practice.

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 116–132, 2013.

Clearly, it is essential to establish complete and consistent requirement and specifications if a product is to have any chance at meeting it intended use needs. While a quality process is essential to developing a quality product, practices within this process can have a direct bearing on the outcome. Over past decades many technologies have been developed, ranging from structured design and object oriented design, to formal methods based design. Interest in mathematically based design has been a constant and much progress has been made in facilitating development in a practicable manner.

However, in reality, formal methods cannot marry well-established industrial process due to lack of the affordability and efficiency to handle a large scale of specification and proof. The divorce between practical software processes and formal methods has been extensively investigated in the past decades.

Formal engineering method (FEM) [1] has been proposed to bridge the gap between software engineering and formal methods. FME addresses the issue of adapting formal methods for industrial software process so that software engineers can easily grasp formal methods and, at the same time, they do not lose the power of the mathematical support. In this paper, we apply the Structured-Object-based-Formal Language (SOFL) [2] to a case study of the Generic Insulin Infusion Pump (GIIP) [3]. SOFL targets on the unification of mathematical notations and industrial software processes via the application of structured method for requirements analysis and specification and an object-oriented approach for design and implementation.

The software design of a generic insulin infusion pump has been illustrated as a case study to demonstrate how a medical device-based software system can be designed, implemented, and finally reviewed [4]. In the medical device industry, each manufacturer not only has its own requirements on a product such as an insulin infusion pump but also meets regulatory requirements such as safety requirements imposed by FDA. Otherwise, the product cannot be approved for the market. In this case study, we will concentrate on the basal management, part of the GIIP system, to demonstrate how SOFL can be applied to design a software system for a medical device. With the application of SOFL methodology, we help illustrate how a medical device-based software system can be leveraged in terms of design, development, validation and finally regulatory review.

The paper is organized as follows. Section 2 introduces SOFL and GIIP application. A software design based on SOFL for GIIP application is given in Section 3. We draw a conclusion in Section 4.

2 Preliminary

In this section, we briefly introduce SOFL, and then describe the GIIP model.

2.1 Introduction to SOFL

The SOFL is a formal framework that unites formal methods with industrial software development processes. In general, SOFL establishes a structured way to specify the requirements of a software system using an object-oriented approach for subsequent

design and implementation based on these requirements. Formal methods can be applied across the entire SOFL-driven development process, to assure high quality specifications and verification at different levels of the intended software system. For example, SOFL allows software engineers to reason about the completeness of software specifications.

A SOFL specification includes a hierarchical condition data flow diagram (CDFD) that links a hierarchy of *specification modules* together. A CDFD is a *directed graph* consisting of *data flows*, *data stores* and *condition processes*. A CDFD describes the static interfaces between components and the dynamical interaction between these components and corresponding data flow. Figure 1 illustrates the basic components of CDFD. The *condition process* is specified with a pre- and post-condition. A *data flow* descriptor identifies how data is exchanged between condition processes. A *data store* defines a variable of a specific type. The *specification module* (s-module) describes the precise functionality of the condition processes in terms of their inputs and outputs. The *s-module* also provides a static definition of all components and details of the system in a textual form.

Fig. 1. CDFD Components

2.2 Introduction to GIIP

The Generic Insulin Infusion Pump (GIIP) (safety) model [5] was developed by FDA to be an open system research platform that establishes safety properties generic to insulin infusion pumps. It was envisioned that academics and manufacturers would experiment with the model and share improvements on its design details and experiment with it to help establish new or improved innovative development technologies. All the requirements of the system can be attributed to two categories: functional requirements given in the GIIP Functional Specification Document [6] and safety requirements given in the Safety Document [7].

The software design of GIIP allows a user to program a time period and an insulin infusion rate so a patient receives the administration of insulin via an insulin infusion pump. Based on the specification [6], the software model of GIIP consists of three primary functional modules: delivery control logic, time management, and interface to User Interface (UI) devices. The design of the system concentrates on the delivery control logic module, which includes several major components as illustrated in Figure 2. The Delivery Control Logic is composed of Basal Management, Bolus Management, Pump Delivery Mechanism Interface (PDMI), and Alarm Handler. Also, all relevant events are recorded via Data/Event Logging (DEL).

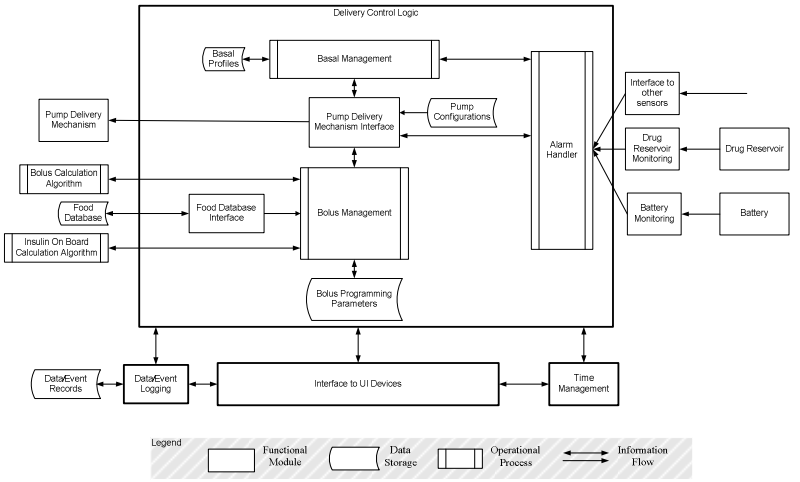

Fig. 2. GIIP Architecture

In this paper, we focus our design on the Basal Management component of the GIIP system. The basal management component is to allow a user/patient to program different insulin infusion rates within 24 hours in a day. Each insulin infuse rate should be given by an effective period– the start time, end time– and the corresponding basal rate, called a *segment*. All insulin infuse rates of a day are programmed to a file, called a *Basal Profile*.

However, sometimes a patient may require special administration of insulin due to some reasons. In this case, the component provides a user with a mechanism to program a high-priority temporary profile, called a *Temporary Basal*, which consists of the duration time and the basal rate. In summary, the basal management component should accomplish two major functionalities: 1) manage basal profiles, and 2) produce the correct information such as the insulin infuse rate based on (normal/temporary) basal profiles to the corresponding component [6].

2.3 GIIP Safety Requirements

The objective of developing the GIIP system is to assure its compliance to a set of core safety requirements, which are articulated to mitigate previous insulin pump failures and other significant safety issues [7]. Throughout this paper, we consider several safety requirements from [7] that govern safe basal administration in GIIP. For the convenience of readers, we reiterate these requirements as follows:

Safety Requirement 1: The pump shall allow the user to program a basal profile with a set of basal rates, ranging from 0.05 to x Units/hour.

Safety Requirement 2: For each basal rate in the profile, the user shall define the duration of the particular rate.

Safety Requirement 3: The pump shall allow the user to set at least two basal profiles at the same time, and require the user to activate no more than one profile at any single point in time.

Safety Requirement 4: The programmed infusion rate of a temporary basal shall not exceed x Units/hour and the duration of a temporary basal shall not exceed y hours.

Safety Requirement 5: The pump shall allow a user to stop a temporary basal while it is in administration.

3 Design of Basal Management Component

In this section, we first outline the structure of the Basal Management component, and then explain in detail how its design is refined to lower levels in a top-down style.

3.1 Top Level Design

In general, the Basal Management component takes the responsibility of managing basal administration according to requests received from the user. The user is allowed to send the component two types of requests regarding basal administration: 1) the BP-req requests, which allow the user to program and manage a basal profile, and 2) the TB-req requests, which allow the user to program and manipulate a temporary basal. Based on such requests, the component decides the current basal rate and outputs it to the insulin delivery mechanism (abstracted as the PDMI component) of the system for delivering insulin. The component also needs to log any changes made to basal administration and report them to the event logging mechanism in the system (abstracted as the DEL component). Figure 3 summarizes, in the format of a CDFD, the interaction that the Basal Management component has with other components in order to fulfill its functionalities.

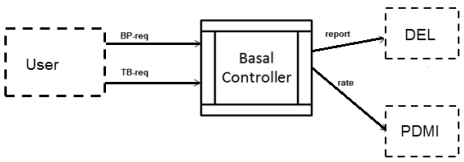

Fig. 3. Top Level CDFD for Basal Management

In Figure 3, a process, Basal Controller, is introduced to represent the Basal Management component. Textual SOFL specifications, as illustrated in Figure 4, are embedded in this process to explicitly define the functionalities of the Basal Management component. As shown in Figure 4, the textual specifications of a process include declaration of variables and data types to be used by the process and its functionalities specified in an object-oriented style.

The user manages basal administration mainly by instructing the Basal Management component to manipulate basal profiles. In particular, the user can request the component to add, delete, update, activate, or deactivate a selected basal profile. To manipulate requests from the user, the component declares an enumeration data type, *BasalProfile-request*, to define the types of operation on basal profiles (see line 2 in Figure 4); and a variable of this type, *BP-req*, to store such request(s) from the user (see line 12 in Figure 4).

```
module Basal Controller
1  type
2       BasalProfile-request = {<ADD>, <REMOVE>, <DELETE>, <ACTIVATE>, <DEACTIVATE>}
3       TemporaryBasal-request = {<SET TB>, <STOP TB>} /* request types */
4       BasalRate = real;
5       SysTime = nat*nat*nat /* hour, minute, second */
6       STime = nat*nat*nat*nat /*date, hour, minute, second*/
7       Report      =           composed of
8                               Index: nat
9                               Type: BasalProfile-request or TemporaryBasal-request
10                              Time:STime
11                              end;
12 var
13      BP-req: BasalProfile-request;
14      TB- req: TemporaryBasal-request;

15 c-process      User()BP-req: BasalProfile-request| TB-req: TemporaryBasal-request
16 post           true
17 end-process

18 process        BasalController(BP-req:BasalProfile-request|TB-req:TemporaryBasal-
                  request)report:Report|rate:BasalRate|dummy:void
19 pre            true
20 decomposition          BasalController-Decom1, BasalController-Decom2
21 comment
22                decompose the normal basal request CDFD to BasalController-Decom1 and the temporary
                  basal request to BasalController-Decom2
23 end-process;
24 process        DEL(report:Report)
25 pre            …
26 post           …
27 end-process;
28 process        PDMI(rate:BasalRate)
29 pre            …
30 post           …
31 end-process;
```

Fig. 4. Module of Top Level CDFD

With regard to managing temporary basal, the user can instruct the component to either start or stop a temporary basal. Thus, the Basal Controller process declares an enumerate data type *TemporaryBasal-request* , which consists of two possible values *SET TB* and *STOP TB*, to represent the operations on temporary basal administration. A variable *TB-req*, with the type *TemporaryBasal-request*, is declared to record the request(s) from the user regarding temporary basal manipulation.

As aforementioned, any changes to the basal administration, normal or temporary, need to be logged. In Figure 4, lines 7-11, a data type *Report* is declared for such logs. The *Report* type consists of three fields: field *Index* corresponds to the unique index number of the basal profile being affected by the change; field *type* indicates what type of change, *BasalProfile-request* or *TemporaryBasal-request*, happens on the selected basal profile; and field *Time* records the exact time when the change occurs. Notably, field *Time* has the type of *STime*, a quadruple recording the date, hour, minute, and second elements of the time.

3.2 Basal Profile Requests (BP-Req)

It is worth noting that, the textual specifications of a process also needs to define the expected way of how the process's functionalities are decomposed, if the complexity of these functionalities justifies further refinement. Take the Basal Controller process for instance. The functionalities of this process can be generally decomposed into those for normal basal management and for temporal basal manipulation. As shown in the second part of Figure 4[1], lines 20-21 explicitly define such decomposition. We first explain in this section how a lower-level CDFD is designed for managing normal basal profiles, and then explain that for temporary basal in section 3.3.

Figure 5 enumerates variables and data types used in managing normal basal administration. Firstly, a data type, Profile, is declared for basal profiles. The *Profile* type is composed of a set of segments, each of which has the type *Segment*, A basal profile with the type Profile distinguishes itself from others with a unique index number, stored in its *key* field.

Each segment in a basal profile is a combination of its effective period (field *EffectivePeriod*) and the associated basal rate (field *basalrate*), where the effective period is defined as the start and end time of the period. Type *EffectivePeriod* is thus declared as a production of two *Systime*-typed elements. Note that type *Systime* is different from *STime* in that the former has only three fields for hour, minute and second elements, while the latter has an extra field for date.

In the system, each basal profile stored is assigned with a unique index number, through which this profile can be fetched, edited, and removed. So, we define type *Profiles* as a *set* of *profile* at line 11 and type *Index* as nature number (denoted as *nat*) to represent the index of a profile at line 12. The basal profiles in the system are stored in variable *profiles,* which is declared at line 39 with type *Profiles.* To map an index number to the corresponding basal profile, a data type, *ProfilesRecord*, is declared at line 13 A a variable *profiles-record* typed as *ProfilesRecord*, is declared at line 34 to store the basal profile fetched based on a user-indicated index number.

With regard to basal delivery, the system takes one of two possible modes at any point of time: Delivery and No Delivery. Thus, line 13 of Figure 5 declares an enumeration type *DeliveryMode* with two values*: Delivery* and *NoDelivery*, and line 35 declares a variable *mode* to store the system's current delivery mode, with the type of *DeliveryMode*.

To eliminate the possible confusion in basal administration, safety requirement 3 enforces that no more than one basal profile be activated at any single point of time. To implement this requirement, we define the variable *activeprofileindex* (line 36 in Figure 5) to maintain the index of the basal profile currently being activated. Apparently, *activeprofileindex* can take only one value at any point of time.

A temporary basal is defined by its duration and its associated temporary basal rate. The data type *TemporaryBasal*, a record type, defined at lines 24-27 of Figure 5, is introduced to represent this fact. The *TemporaryBasal* has a *nat*-typed field

[1] The second part of Figure 4 also includes declaration for the *DEL* and *PDMI* components. Since these two components are not the focus of this paper, we skip their details here.

```
module BasalController-Decom1 / Basal Controller
1  type
2  Profile =        composed of
3                   segments: set of Segment
4                   key: nat
5                   end;
6  Segment =        composed of
7                   effectiveperiod: EffectivePeriod
8                   basalrate: real
9                   end; /* Based on the requirement of safety 2*/
10 EffectivePeriod = Systime * Systime ; /*start time, end time */
11 Profiles = set of Profile
12 Index = nat;
13 ProfilesRecord = map Index to Profiles;
14 DeliveryMode = {<Delivery>, <NoDelivery>}
15 Result = bool;
16 AddRequest    =          Profile;
17 DeleteRequest =          nat /* index of profile in the list*/
18 UpdateRequest=           composed of
19                          index: nat
20                          profile: Profile
21                          end;
22 ActivateRequest =        nat /*index of profile in the list*/
23 DeactivateRequest =      void;
24 TemporaryBasal =         composed of
25                          duration: nat /* in hour */
26                          rate: real
27                          end;
28 var
29 add-req: AddRequest;      /* an instance of Add request from user*/
30 del-req: DeleteRequest;   /* an instance of Delete request from user*/
31 upd-req: UpdateRequest;   /* an instance of Update request from user*/
32 act-req: ActivateRequest; /* an instance of Activate request from user*/
33 deact-req: DeactivateRequest;  /*an instance of Deactivate request from user*/
34 profiles-record: ProfilesRecord;
35 mode: DeliveryMode;      /* the delivery mode of the basal management component*/
36 activeprofileindex: nat;    /* represent the index of active profile in the profile list */
37 tempbasal:TemporaryBasal; /* an instance of temporary basal */
38 index:Index
39 profiles: Profiles
40 currentTime: Systime      /* current system time
```

Fig. 5. Type and Variable Declarations for BP Request

duration to define the duration of a temporary basal in hours, and the *real*-typed field *rate* documents its temporary basal rate (in Unit/Hour). Any temporary basal input by the user is stored in the variable *tempbasal* (line 37 in Figure 5), the type of which is *TemporaryBasal*. The current system time is represented by variable *currentTime,* whose type is *SysTime* given at line 40.

The *BP-Req* type of requests can be further refined to the following five types [6], based on what action the user intends to perform on the basal profiles:

- Request to add a basal profile (denoted as AddBasal)
- Request to delete an existing basal profile (denoted as DeleteBasal)
- Request to update an existing basal profile (denoted as UpdateBasal)
- Request to activate an existing basal profile (denoted as ActivateBasal)
- Request to deactivate the currently activated basal profile (denoted as Deactivate-Basal)

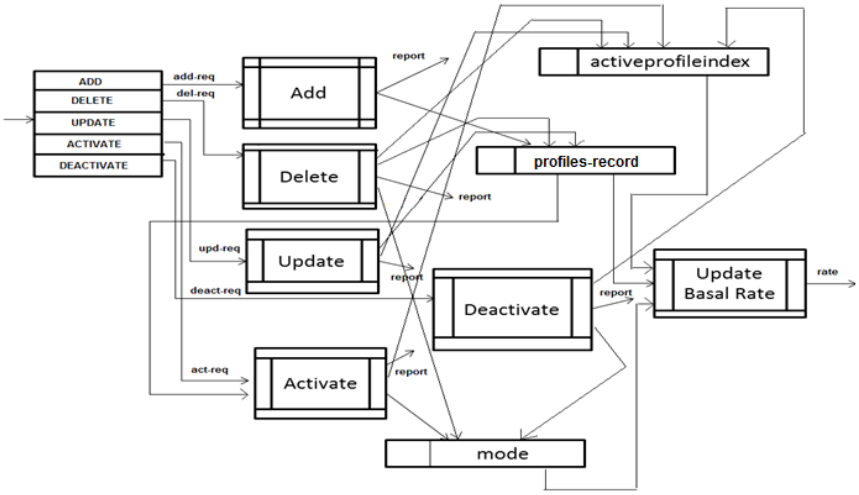

Fig. 6. Decomposition of Basal Controller (BP-Request, Partial)

These five types of requests are declared from line 16 to line 23 in Figure 5. Note that AddBasal requests can be distinguished by the new basal profile(s) they address. Thus, we declare the data type *AddRequest* as equivalent to type *Profile*. The actual AddBasal requests received from the user are stored by variable *add-req*.

DeleteBasal requests, on the other hand, refer to the basal profiles to be deleted by their index numbers. Thus, we declare type *DeleteRequest* as *nat*, while all DeleteBasal requests are stored in variable *del-req*, which is declared at line 30 of Figure 5.

UpdateBasal requests require a user to provide the index of the profile to be updated, as well as the new profile to replace it. Therefore, a record type *UpdateRequest* is declared at line 15 of Figure 5 representing such a request, which consists of two fields: a *nat*-typed field *index* for the index, and a *Profile*-typed field *profile* for the new profile. In addition, all UpdateBasal requests from the user are stored in variable *upd-req*.

ActivateBasal requests are similar to DeleteBasal requests. Thus, a type *ActivateRequest* is declared for these requests, which is also *nat* to indicate the index of the basal profile under concern. ActivateBasal requests are stored in variable *act-req*.

DeactivateBasal requests do not require any additional parameters. Thus, the type *DeactivateRequest* for these requests is declared as a void type. A variable *deact-req* with type *DeactivateRequest* is declared to store DeactivateBasal requests from the user.

In our GIIP design, the *Basal Controller* component manipulates *BP-Req* requests from the user based on the types of such requests. That is to say, the behavior of the component is decomposed several subsets, each of which corresponding to a particular type of *BP-Req* requests. The CDFD in Figure 6 illustrates such decomposition. Moreover, a process is defined for manipulating each type of requests. The rest of this section discusses the details of all these processes.

Figure 7 depicts the process for handling AddBasal requests. The user can create multiple basal profiles that, if valid, are stored in the basal profiles record in the system. A basal profile is valid if it complies with safety requirements 1-3. Thus, the process in Figure 7 first validates whether an input profile against these safety requirements using the precondition at line 5. The precondition calls the method *validate,* which is defined by lines 16-22. The *validate* method first compares the start time (parameter 1 of *effectiveperiod*) and end time (parameter 2 of *effectiveperiod*) of the effective period of each segment in the basal profile under concern. If the former is before the later, then the method checks if the basal rate associated with this segement is greater than 0 Unit/Hour and less than a thereshold specified by the user (denoted as *X* in Figure 7). Any basal profile failed in these checks is considered as invalid and will be discarded.

If an input profile is valid, the process adds it into the profiles record through the following steps: 1) Add the new basal profile to the set *profiles* at line 8. 2) Point the current index to the new profile and override the previous profiles record using the keyword *override*; 3) Update the index of the next profile by increasing *index* by 1, as shown at line 10, and 4) generate a report on this action and send it to the DEL component.

```
1  process Add(add-req:AddRequest|result:bool) report:Report|dummy:void
2  ext     wr profiles-record: ProfilesRecord
3          wr index: Index
4          wr profiles: Profiles
5  pre     validate(add-req)
6  post
7          add-req.key = index
8          profiles = profiles + add-req
9          profiles-record = override (~profiles-record, {index -> add-req})
10         index = index + 1
11         bound(report)
12 comment
13         if the input profile pass the validation process, add new profile to the profile list and
14         generate a report. otherwise, do nothing.
15 end-process;
16 function validate(profile: Profile)re: bool
17         pre         true
18         post        if not exists[ segment inset add-req.segments | segment.effectiveperiod(1) >
           segment.effectiveperiod(2) and segment.basalrate > X]
19                     then re = ture
20                     else re = false
21 comment           validate the input profile is valid or not
22 end-function;
```

Fig. 7. Module for Add Profile

Figure 8 shows the process responding to DeleteBasal requests. This process first checks the presence of the profile to be deleted in the *profiles* record, as enforced by the pre-condition at line 6. If the profile does not exist in the *profiles* record, the process simply discards the request. Otherwise, it locates the index of the profile and removes it from the record (line 8). The process, like others, also generates a report if a profile is deleted, and sends the report to the DEL component.

Furthermore, if a request of deleting one basal profile is valid, and the profile to be deleted is currently the active profile (i.e., *index* of the request equals to

acitveprofileindex), then the component needs to conduct the following tasks (as shown from line 9 to line 12 in Figure 8):

1. Deactivate the profile to be deleted without activating another one (i.e., setting the *acitveprofileindex* to -1);
2. If there is no temporary basal currently in process, set the delivery mode to No Delivery, indicating that there is no basal, in any form, currently under administration.

```
1  process Delete(del-req: DeleteRequest)report:Report and
2  ext     wr profiles-record: ProfilesRecord
3          wr activeprofileindex: nat
4          wr mode: DeliveryMode
5          rd tempbasal: TemporaryBasal
6  pre     exists[ r inset dom (profiles-record) | r = del-req]
7  post    let pfile: rng(profiles-record) | pfile.key = del-req in
8          profiles-record = rngdl (~profiles-record, {pfile}) and domdl ({r}, profiles-record)
9          if del-req = activeprofileindex
10         then activeprofileindex = -1 and
11         if tempbasal = nil and activeprofileindex = -1
12         then mode = NoDelivery
13         bound(report)
14 comment         remove the selected profile from profile list and switch delivery mode if necessary
15 end-process
```

Fig. 8. Module for Delete Profile

The *Update* process shown in Figure 9 is defined to manipulate UpdateBasal Requests. Similar to handling DeleteBasal requests, this process first checks if a basal profile with the index indicated by the user exists in the *profiles* record. If not, the process simply ignores the request. Otherwise, the process validates the new basal

```
1  process Update(upd-req:UpdateRequest)report:Report|dummy:void
2  ext     wr profiles-record: ProfilesRecord
3          wr activeprofileindex: nat
4          wr profiles: Profiles
5  pre     exists[ r inset dom (profiles-record) | r = upd-req.index] and
           validate(upd-req.profile)
6  post    let x: dom (profiles-record) | x = upd-req.index in
7          let pfile: rng (profiles-record) | pfile.key= upd-req.index in
8          profiles-record = rngdl (~profiles-record, {pfile})
9          profiles = profiles + upd-req.profile
10         profiles-record = override (~profiles-record, {x -> ~profiles-record(x)})
11         bound(report)
12 comment
13         if the input profile pass the validate process, update new profile to the profile list and
14          generate a report
15         otherwise, do nothing.
16 end-process;
```

Fig. 9. Module for Update Profile

profile by calling the *validate* method. If the new basal profile is valid, the process replaces the previous profile in the *profiles* record that has the user-specified index number with the new basal profile. If this happens, a report is generated and sent to the DEL component for logging.

Since the profiles record may contain multiple profiles, a user can activate one of them and use it to decide the output basal rate. The *Activate* process, as illustrated in Figure 10, is crafted to assist the user in dosing so. This process first checks whether or not the selected basal profile, i.e., parameter *act_req*, exists in the profiles record (line 5). If yes, the process updates the active profile variable *activeprofileindex* to the index of the selected profile, *act-req,*(line 6). After this, the system switches to the *Delivery* mode (line 7), and generates a log and feeds it to the DEL (line 8).

```
1  process Activate(act-req:ActivateRequest)report:Report
2   ext    wr activeprofileindex: nat
3          wr mode:DeliveryMode
4          rd profiles-record: ProfilesRecord
5   pre    exists [pfile inset rng (profiles-record) | act-req = pfile.key]
6   post   activeprofileindex = act-req
7          mode = "Delivery"
8          bound(report)
9   comment
10         set the active profile.
11 end-process
```

Fig. 10. Module for Activate Profile

The system allows a user to deactivate an active profile, a feature implemented by the *Deactivate* process in Figure 11. This process first voids the *activeprofileindex* variable by settings it to -1 (line 6) , and then, if there is no temporary basal defined, switches to the *No delivery* mode (line 8). The entire process is recorded in a log sent to the DEL (line 9).

```
1  precess Deactivate(decatc-req:DeactivateRequest)report:Report
2   ext    wr activeprofileindex:nat;
3          wr mode:DeliveryMode
4          rd tempbasal:TemporaryBasal
5   pre    true
6   post   activeprofileindex = -1
7          if tempbasal = nil
8          then mode = "NoDelivery"
9          bound(report)
10  comment
11         Switch the delivery mode based on the safety requirement
12  end-process
```

Fig. 11. Module for Deactivate Basal Rate

In order to decide the actual basal rate, we define the *UpdateBasalRate* process that calculates the actual basal rate continuously during the system execution. The process is shown in Figure 12 based on the following rules:

1. If the delivery mode is *NoDelivery,* the basal rate is 0;
2. Otherwise, if there is a temporary basal in progress, then the basal rate is set as the temporary basal rate;
3. Otherwise, the process should find the active profile (if any) from the *profiles* record, decide the right segment in the active profile that covers the current time, and use the basal rate associated with this segment as the actual basal rate.

```
1  process UpdateBasalRate()rate:BasalRate
2  ext     rd profiles-record: ProfilesRecord
3          rd tempbasal:TemporaryBasal
4          rd activeprofileindex: nat
5          rd mode:DeliveryMode\
6          rd currentTime
7  pre     true
8  post    if mode = "NoDelivery"
9          then rate = 0;
10         else if tempbasal != nil
11         then rate = tempbasal.rate
12         else if activeprofileindex = 0
13         then rate = 0
14         else let pfile: rng(profiles-record) | pfile.key = activeprofileindex
15         rate = getrate(pfile)
16         bound(rate)
17         comment
18         Calculate the basal rate and send the rate to PDMI component
19 end-process
20 function getrate(pfile: Profile) re: basalrate
21         if exists[ segment inset pfile.segments | segment.effectiveperiod(1) <= currentTime and
           segment.effectiveperiod(2) > currentTime]
22         then       re = segment.basalrate
23 end-function
```

Fig. 12. Module for Update Basal Rate

3.3 Temporary Basal Requests (TB-Req)

A user can instruct the system to start or stop a temporal basal. In particular, the user can send to the system a *SET TB* request for programming and start a temporary basal. Or he/she can request to stop the currently ongoing temporary basal with *STOP TB* requests. In particular, a *SET TB* request carries a temporal basal (duration and temporary rate) as the parameter, while a STOP TB request has no parameter.

Figure 13 illustrates how the GIIP system should respond to these types of request, where a Set TB process is introduce to manipulate *SET TB* requests and a Stop TB process to manipulate *STOP TB* requests.

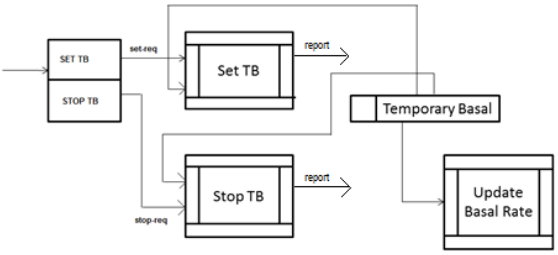

Fig. 13. Basal Controller Design for Temporary Basal Requests

Figure 14 provides type and variable declarations for the GIIP design with regard to managing temporary basal requests, where the types *SetTBRequest* and *StopTBReques* are defined for the two types of requests related to temporary basal, respectively. Note that type *StopTBRequest* is actually a void type, as it does not require any parameter to stop the current temporary basal. Variables *set-req* and *stop-req* are declared to stop the temporary basal requests from the user.

```
module BasalController-Decom2 / BasalController
1  type
2  SetTBRequest    =        TemporaryBasal;
3  StopTBRequest   =        void;
4  TemporaryBasal  =        composed of
5                           duration: nat /* in hour */
6                           rate: real
7  var
8   set-req: SetTBRequest;      /* an instance of Set temporary basal request from user */
9   stop-req:StopTBRequest;     /* an instance of Stop temporary basal request form user */
10 profiles-record: ProfilesRecord;
11 mode: DeliveryMode;
12 tempbasal:TemporaryBasal; /* store the temporary basal */
```

Fig. 14. Type and Variable Declarations for TB Request

A process SetTB, as shown in Figure 15, is defined to specify the process for the GIIP to respond to a SET TB request from the user. As imposed by safety requirement 4, any new temporary basal that the user intends to initial, should have a temporary basal rate not greater than x Units/hour and a duration not greater than y hours (both x and y are thresholds pre-specified by the user). The *SetTB* process first checks to assure that no other temporary basal is currently stored in the system (see line 3 of Figure 15), and then checks whether the input temporary basal is valid by calling the *validateTB* method. The *validateTB* method, defined from line 8 to line 12 in Figure 15, checks whether or not the configuration of the input temporary basal is within the ranges prescribed in safety requirement 4. . If the input temporary basal is valid, the component stores it in variable *tempbasal*, generates a report indicating that the input temporary basal is initiated, and feeds the report to the DEL componet . Once variable *tempbasal* is set, the *UpdateBasalRate* process will update the output basal rate according to the temporary basal.

```
process SetTB(set-req:SetTBRequest)report:Report
ext     wr tempbasal:TemporaryBasal
pre     tempbasal = nil and
        validateTB(set-req)
post    tempbasal = set-req
end-process
function validateTB(tempbasal:TemporaryBasal)re : bool
pre     ture
post    if 0.05<= tempbasal.rate and tempbasal.rate < X and
        tempbasal.duration < Y
        then re = true
        else re = false
        bound(report)
comment
        Validates the temporary basal based on safety requirement 4
end-function
```

Fig. 15. Module for SetTB

In terms of stopping the current temporary basal, the *StopTB* process, shown in Figure 16, simply clears variable *tempbasal* by setting it to *nil*.

```
process StopTB(stop-req:StopTBRequest)report:Report
ext     wr          tempbasal:TemporaryBasal
pre     ture
post    tempbasal = nil
        bound(report)
end-process
comment         allow the user to stop the temporary basal(safety 5)
```

Fig. 16. Module for Stop TB Request

4 Related Work

To assure the correctness of safety-critical software systems, many formal methods based approaches have been proposed in last few decades, including Alloy [8], ASM [9], B [10], and Z [11]. All of these approaches are built on solid mathematical foundation. Although such solid mathematical foundation enable formal verification to be conducted on software systems thus developed, it also restricts the applicability of these approaches in industrial development practices. That is to say, developers who intend to take advantage of these approaches, they have to first become familiar with the mathematical foundation underlying them. Consequently, the extra learning curve and sophisticate mathematical background hinder engineers from applying these approaches in real industrial practices.

5 Conclusion

The application of formal methods in industrial practices has been hurdled by both the steep learning curve to master these methods and computational expressiveness underlying these methods. The SOFL methodology intends to overcome these hurdles

by integrating together a formal representation framework and an object-oriented development process. It has been proven as effective when applied to various applications [12].

In this paper, we applied the SOFL methodology to develop the Generic Insulin Infusion Pump design. While concentrating on the Basal Management component in the GIIP system, our work can easily be extended to the rest of the GIIP system. More importantly, this case study helps us understand how SOFL leverages the development of a complex software system, such as medical device software, from the following aspects:

1. A software development process driven by the SOFL helps to capture the traceability [13] among different software artifacts, such as SOFL specifications and SOFL implementation modules. Good quality traceability information can greatly improve the correctness and maintainability of complex software systems. For example, the missing trace information from a requirement to a design element indicates that the requirement is more likely missed in the system [14]. Moreover, good traceability information can help third-party reviewers understand such systems with less effort and more accuracy.
2. The formal and practical aspects of the SOFL methodology introduce more rigorousness to the development of complex software systems. Formal verification can be applied to inspect the correctness and consistency of SOFL specifications for these systems, which in turn helps developers as well as third-party reviewers establish higher confidence in these systems.

As the future work, we plan to apply the SOFL methodology to the rest of the GIIP system, and to investigate effective verification techniques for inspecting SOFL specifications.

Acknowledgements. The authors would like to thank Drs. Yi Zhang and Paul Jones at FDA who provided valuable and helpful suggestions about the GIIP project, and the great comments and feedback about this paper. Last, the first author appreciates their guidance of the GIIP project during his internship in summer 2011.

References

[1] Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer (2004) ISBN 3-540-20602-7
[2] Liu, S., Offutt, J., Ho-Stuart, C., Sun, Y., Ohba, M.: SOFL: A Formal Engineering Methodology for Industrial Applications. IEEE Transactions on Software Engineering 24(1), 24–45 (1998)
[3] Zhang, Y., Jones, P., Jetley, R.: A Hazard Analysis for a Generic Insulin Infusion Pump. Diabetes Science and Technology 4(2) (2010)
[4] Vogel, D.: Medical Device Software Verification, Validation, and Compliance. Artech House (2011)
[5] Generic Infusion Pump Project, `http://rtg.cis.upenn.edu/gip.php3`
[6] FDA, GIIP Functional Specifications (2011)

[7] Zhang, Y., Jetly, R., Jones, P., Ray, A.: Generic Safety Requirements for Developing Safe Insulin Pump Software. Diabetes Science and Technology 5(6), 1403–1419 (2011)
[8] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006) ISBN 978-0-262-10114-1
[9] Gurevich, Y.: Evolving Algebras. In: Specification and Validation Methods, pp. 9–36. Oxford University Press (1995) ISBN 0-521-49619-5
[10] Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996) ISBN 0-521-49619-5
[11] Schuman, S.A., Meyer, B., Abrial, J.-R.: A Specification Language. In: McKeag, R.M., Macnaghten, A.M. (eds.) On the Construction of Programs. Cambridge University Press (1980)
[12] Liu, S., Stavridou, V., Dutertre, B.: The Practice of Formal Methods in Safety Critical Systems. Journal of Systems and Software 28(1), 77–87 (1995)
[13] Spanoudakis, G., Zisman, A.: Software Traceability: A Roadmap. In: Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing (2004)
[14] Yadla, S., Huffman Hayes, J., Dekhtyar, A.: Tracing Requirements to Defect Reports: An Application of Information Retrieval Techniques. Innovations in Systems and Software Engineering: A NASA Journal 1, 116–124 (2005)

Extension on Transactional Remote Services in SOFL

Yisheng Wang and Haopeng Chen

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

Abstract. Software quality always attracts considerable attentions of people. Software running without any mistakes is always a dream of all developers. Besides traditional testing method using in practice such as path coverage, selection coverage, etc, people try to use some more formal and reliably method to ensure the quality. SOFL, stands for Structured Object-oriented Formal Language, is a kind of formal language which can be used to describe, validate and verify core business flow of software. As software developing model keeps changing for years, we need to make some extensions to SOFL. In this paper, we have performed extension on transactional remote services designed for SOFL. Our extension can mainly be divided into two parts: remote services and transactions. By introducing these, SOFL is able to keep pace with the changing software developing model, thus ensuring software quality in a more mathematical and different way comparing with traditional testing.

1 Introduction

A mature and practical commercial software product always needs a relatively long period of time and cooperation of many people including managers, designers, developers and testers. Nevertheless, software quality still cannot be perfectly guaranteed. Bugs and maintaining costs are bottlenecks of software industry to some extent[1].Theoretically, each software product contains potential problems and whether it would crash in the next second remains unknown[2]. People have already found a lot of ways to improve software quality such as standard developing processes and software testing methods. Using these classic methods such as RUP developing process[3], UML[4], black-box testing and white-box testing can improve software quality, but still 100% correct is unable to be reached or proved.

Actually, there are some other researches focusing on formal methods in software developing. SOFL[5] is a kind of formal language which can be used to describe, validate and verify a business workflow in a software product. Usually we use SOFL Specification and CDFD (stands for Control and Data Flow Diagram) to model a workflow. These can still be divided into some basic elements such as Module, Process, Dataflow, Datastore, etc. In general CDFD describes

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 133–147, 2013.

relationships of these elements such as dataflow connecting two neighboring processes. It should be noted that there also exists hierarchical relationship between two CDFD Diagrams. SOFL Specification carries out the detailed implementation of a module, including its pre and post condition, data store, input and output variables. It plays a complementary role together with CDFD.

After modeling a workflow using SOFL, we are able to find whether there are some potential mistakes in the model by validating and verifying which will be described afterward. The correctness of a workflow validated and verified by SOFL can be ensured by formal methods[6][7].

The programming model of software product always keeps changing. Invoking remote services helps developers to reuse codes as well as avoid duplicating time-consuming developing job, which is not supported or introduced in SOFL. Moreover, transaction is an important part in software product, especially for web-based applications with relatively complicated business logic. If SOFL can keep pace with this changing trend, its influential range will spread, thus promote fames of software formalization. This is also what our extensions aim to.

The rest of the paper is organized as follows: Section 2 lists some motivations of our work. Section 3 introduces the detailed extensions we have performed, dealing with remote services and transactions individually. Section 4 summarizes a practical case modeling with SOFL along with our extensions. Section 5 introduces related work by others on SOFL recently. Section 6 summarizes the main contribution of this paper and comments on further research.

2 Motivation

As the day passed, software developing model has changed a lot comparing with several tens of years ago. What software developers focus on now is to reuse as much existing remote services as possible rather than writing and testing same and duplicate codes. This is also the sharing notion of both component-based software engineering[8] and SOA[9]. The difference between component and service mainly lies in the location of the reusing piece of codes. Components can be withdrawn and run locally while services can not. In general case, we invoke remote services by providing input parameters and waiting for output results through network connections. Its running process is totally different comparing with local processes. Moreover, as Cloud Computing gaining more and more attentions of people, this notion has been inherited and carried forward by invoking remote services in Cloud[10].

In SOFL, there are several concepts dealing with 'external' elements. 'External Process' means a virtual process situated before the first process or after the last process. Generally, external processes are used to describe the end user or third-party system in a workflow. Users providing input data or operations and third-party system which receives output data from the workflow can be called external processes towards it. Another concept reads 'external stores' which is a kind of data store. It is used to describe 'external devices or files, such as displays, files on disks, printers, keyboards'[5]. All these seems to be not very

related with the scenario of reusing. Unfortunately, support on these aspects of SOFL is quite weak. As SOA and Cloud Computing boosts in a fantastic speed, SOFL should also keep its pace with these popular notions.

Another key point towards web-based application, especially dealing with online payment or finance elements is transaction. A well-informed instance is saving, withdrawing or transferring money from a bank, which we can simply find in our database textbook. More importantly, as the introducing of invoking remote services, the possibility of invoking exceptions increased because of the unreliability of network connections. According to Probability Theory , if more than one remote services or data stores are involved in a workflow, the probability of mistakes will boosts in an exponential speed. It can be believed that transactions are required especially for workflow containing remote services or data stores. We can also inferred from the scenarios that long transactions would play a dominate role because of network delay and business logic. Assuring its correctness is also important towards a practical application. So it is reasonable for us to make extension to SOFL and enable it to deal with transactions, including long transactions.

It is clear that the requirement of extension to SOFL about remote services and transactions is reasonable and practical. We should add remote services, remote data stores and transactions into both SOFL Specification and CDFD based on the existing rules and grammars of SOFL. The detailed information of our extensions is shown in the next section.

3 Approach

According to what we have carried out in previous sections, our extensions on transactional remote services to SOFL can mainly be divided into remote elements and transactions. So we will give out our detailed ideas about these individually in the following.

3.1 Remote Elements

We have discussed the changes of programming model recently. These changes can mainly be concluded to distributed architecture (including distributed programming model such as Hadoop or distributed data store such as HDFS), using or revealing web service APIs, different database structure such as no-SQL database, etc.[11] Meanwhile, the ultimate goal of SOFL is to model and review core workflow of a software product. Thus extendibility of SOFL according to the changes made by software world are essential, otherwise it might be eliminated because of its fogyism.

SOFL is such a formal language that it does not care how much computing resources or its distributing and connecting situation a workflow actually used. This means information like whether distributed architectures are introduced, what kind of database is used cannot be revealed from SOFL. In other words, these information is transparent and inconsequential for modeling and reviewing

because SOFL focuses on business logic more. But remote elements is different. Invoking remote services is actually a part of workflow, or said business process. Their differences over local services mainly lie on the unreliability of network connection and the existence of network delay. The difference between using remote data store and local ones is similar. To conclude, we do not care much about the detailed implementation of services or data stores, but care much about what we used is remote ones or local ones.

In the following chapters, we will discussed remote services and remote data stores. Both their textual and visual appearance will be displayed.

Remote Process. Service is the most basic unit in SOA. Services are well-defined business functionalities that are built as software components (discrete pieces of code and/or data structures) that can be reused for different purposes.[9] Service providers can be anyone who wants to be. They need to publish its interface and access information to the service registry. So such services which are invoked by a workflow and do not run on local node is called 'Remote Services'. By invoking remote services, a workflow need to provide input parameters and receive output results as the access information mentioned through network connection.

In SOFL, local function call is described by using 'process'. A process has its name, input parameters, output parameters, pre-conditions and post-conditions. If it uses data stores, a data flow described by a straight line with an arrow will be added in CDFD. Here pre-condition means the rule that input parameters must obey, and post-condition means the calculating process of results, thus is corresponds to function body.

In most cases, a remote service also have such five elements and their information is published in registry node. We want to point out that function body of remote service is transparent to service resumer in most cases. So the post-condition of remote service should be filled by user according to the describing and expect output information in service registry. The SOFL Specification we designed for remote services (or said remote process in SOFL's definition) is shown in snippets 1.

Algorithm 1. SOFL Specification of Remote Process

```
1 remote process Sample (x: T_i_1, y : T_i_2)z : T_o_1, w : T_o_2
2 pre P(x, y)
3 post Q(x, y, z, w)
4 end_process;
```

We can find that the only difference towards normal process in SOFL is the keyword **remote**. It does not appear in original SOFL grammar, thus conflict and ambiguity will not be caused by adding this. It has been mentioned above, post-condition of remote process should be given out by user when modeling

workflow according to the using scenario and intention of that service. More specific example about how to fill the post-condition of remote process can be seen in the next section.

After giving out the SOFL Specification extension, we continue to perform the extension on CDFD. Its difference towards original process in SOFL is only the letter 'R' in the right top side of the diagram. It can be seen in figure 1.

Fig. 1. CDFD of Remote Process

Besides difference in displaying symbol in CDFD and grammar, there are also some extra constraints of remote process. They are listed as follows.

1. There is an extra and common pre-condition of all remote processes, which is that the remote service must be accessible through network connection. Otherwise it can be regarded as a violation of pre-condition of process, thus leading to failure of the whole workflow.

2. Remote process in CDFD can not be decomposed into child-level further because the principle that detailed implementation of remote service is transparent to service consumer.

3. If this remote process has used data stores which does not belong to the workflow, it should not be displayed in CDFD. But if the data store belongs to the workflow, it still needs to be displayed. For instance, if a remote process needs to modify data saved in local database which is abstracted as a data store in SOFL, it should be displayed in CDFD with an arrow line linking from remote process to it.

Remote Data Store. Remote data store means data stored on remote node such as remote database or storage service. A remote data store is an external data store because its storing location is external towards the workflow. So we need to add mark # when they are declared in a module. To be distinguish from normal data stores, we add keywords **remote** before variables. Thus, the SOFL Specification of a process using remote data stores is shown in snippets 2.

The overall specification grammar is quite similar to normal data store, but different at the keyword **remote**, which has already been defined and applied in remote processes. According to the definition of external stores in SOFL, they are global variables. Therefore, remote data stores are also global variables.

We have also designed the CDFD of process declared in snippets 2. It is shown in figure 2.

Algorithm 2. SOFL Specification of Processes Using Remote Data Store

```
process Sample2 (x: T_i_1, y : T_i_2)z : T_o_1, w : T_o_2
ext
    remote rd #v_1 : Te_1
    remote wr #v_2 : Te_2
pre P(x, y, v_1, v_2)
post Q(x, y, z, w, ~v_2, v_1, v_2)
end_process;
```

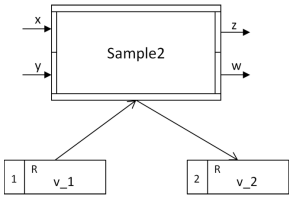

Fig. 2. CDFD of Processes Using Remote Data Store

Similar to CDFD of remote process shown in figure 1, the only different between remote data store and normal data store lies in the **remote** keyword in specification and the 'R' mark in CDFD.

3.2 Transactions

A transaction by definition must be ACID which stands for atomic, consistent, isolated and durable[12]. Usually it provides an "all-or-nothing" proposition, stating that each work-unit performed in a database must either complete in its entirety or have no effect whatsoever. In software products especially web-based applications, transactions are widely used in order to ensure the correct running of the whole system and avoid data inconsistency or business logic chaos. The simplest and most popular example is transferring money between banks. It can be said that most software products might not run correctly without transactions.

SOFL has not introduced transactions according to its original definition. In SOFL, a workflow is running by steps from one process to its successor without operations of opposite direction such as rollback. Thus it is difficult for us to use SOFL to model a workflow with transactions. We have performed our definition of transactions in SOFL in the following paragraphs.

The most basic unit of SOFL-modeled workflow is process. So it is reasonable to introduce transaction based on processes, which means transactions in SOFL are composed with several processes. According to the feature in SOFL that processes can be further decomposed and CDFD is hierarchical, we also want to carry out some constraints on transactions as follows.

1. All processes in a transaction must be neighbored in CDFD. This is also a conventional constraint of transaction.

2. All processes in a transaction must be in a same CDFD. For example, the top level of a workflow is composed with three processes named 'A', 'B', 'C' individually. 'A' can be further divided into 'Aa' and 'Ab'. 'B' is composed with 'Ba', 'Bb' and 'Bc'. According to this rule, 'Aa' and 'Ab' can be in a same transaction, but 'Ab' and 'B' can not composed of a transaction. This is mainly because the inner structure of 'A' is transparent to 'B'. And usually, a process is relatively independent to other process. The fact that a process has several child-level processes is mainly because it has divided its function into several parts. Just focus on the previous instance, the function of 'A' and 'B' is different and independent. It can not be very common that 'Ab' and 'B' need to be in a same transaction. Similarly, 'Ab' and 'Ba' can not be in an transaction either.

If a transaction need a lot of time to finish, we call it long transaction. There is not a very exact and strict boundary time to distinguish long transactions against normal ones. Normal transaction is implemented by executing it in memory and flush data to disk when committed. But such implementation can not be applied directly to long transactions because the performance influence of write-lock and data size. In order to ensure the ACID of long transactions, a series of operation called 'compensating transaction' is introduced and invoked when rollback operation is needed.

In practice, the implementations of compensating transaction is roughly divided into two. The first is writing the inverse operation by user and execute it as rollback. The other is to take snapshots before transaction starts and set all variables to that value when rollback is needed. Its detailed implementation contains the following steps. Before the first process of a transaction is started, a snapshot of the whole system is taken and saved. This snapshot is actually a set of values of current variables of the whole system. Then the transaction started by executing processes it contains in a certain order. When it need to rollback for some reasons, the snapshot is used. We define a set of operations as compensating transaction which sets the value of variable to the original state. It is determined by the snapshot taken before the starting of the transaction. After these set operations has finished, the transaction has rollback to its original state, thus guarantee the "all-or-nothing" feature.

Our extension supports both methods. If user has not written his compensating transaction, taking snapshot is used by default. The selection of rollback strategy is made by user.

There are hierarchical relations among transactions, which means nested transactions are allowed. A remarkable difference between nested transactions against normal ones mainly lies on the retry count. We assume transaction A contains transaction B and we set retry count to n. This means A is failed if and only if B has rollback for n times. In other words, failure of B only once would not cause failure of A if n is greater than 1. Moreover, in general case, nested transactions are long transactions.

We have discussed the definition and some constraints of transactions in SOFL along with long transactions and the implementation of their compensating transactions. We are ready to give out our extension on grammar in SOFL Specification. Because multiple processes can be involved in a transactions, so we need to declare a transaction first, and then refer it in the process definition block if it belongs to this transaction. In addition, because all processes of a transaction must be in a same CDFD, so the declaration of transaction can be placed in the module which is composed with these processes. A sample module with transaction is defined in snippet 3.

In snippet 3, a module named '*Sample_Module*' is defined and indicates '*Sample_Parent_Module*' as its parent module. The general procedure of declaring a module is defined in the following order: constants, types, variables, invariants, behavior and processes. Our extension is to add transactions sections between invariants and behavior and start with the keyword **transaction**. Transaction has the only attribute: name. It is also the only mark when using to distinguish from others, thus it must be unique. In order to show hierarchical structure, '*Trans_2*' behaves as the child transaction of '*Trans_1*' with retry count equaling 3.

There is also an extra transaction definition section in process definition sections just after the name and parameter section. It indicates which transaction it belongs to. Literally, '*Trans_1*' only contains one process, but actually it acts as the parent transaction of '*Trans_2*', thus it contains all these three processes by analyzing the hierarchical relationships. A process can belong to multiple transactions. The transaction section in process definition sections is able to declare all transactions it belongs to by using semicolons as their separators, but ancestor transactions need not to be declared.

Process named '*Sample_Compensating_Process*' is defined as the compensating transaction of '*Trans_2*' by using keyword **compensating transaction**. Note that each process can only declare one transaction name as its compensating transaction. There are not any compensating transactions declaring for '*Trans_1*', thus the default taking snapshots method is used.

Transactions can also be displayed in CDFD using special symbols. We define a dashed box around processes as all of these processes belong to a same transaction. Its name is marked in the dashed box to be distinguished against others. Just take the snippet 3 as an instance, its CDFD is demonstrated in figure 3.

Compensating transaction '*Sample_Compensating_Process*' is not displayed in CDFD because it is not the basic flow of this workflow. Dashed box is shown only in CDFD of this level. It would not display in other level, such as the CDFD for module '*Sample_Parent_Process*' in the example.

4 Evaluation

We have performed our detailed extension method on SOFL about transactional remote services both in CDFD and SOFL Specification in the previous section. In this section, we try to demonstrate a case modeling with extended SOFL to

Algorithm 3. SOFL Specification of Module with Transaction

```
1  module Sample_Module / Sample_Parent_Module
2  transaction
3      Trans_1;
4      Trans_2 / Trans_1 retry 3;
5  behav CDFD_Sample_Module
6  process Sample_Process_1 (x: T_{i_1}, y : T_{i_2})z : T_{o_1}, w : T_{o_2}
7  transaction
8      Trans_2;
9  ext
10     rd v_1 : Te_1
11     wr v_2 : Te_2
12 pre P(x, y, v_1, v_2)
13 post Q(x, y, z, w, ~v_2, v_1, v_2)
14 end_process;
15 process Sample_Process_2 (z: T_{o_1}, w : T_{o_2})v : T_{o_3}
16 transaction
17     Trans_2;
18 ext rd v_2 : Te_2
19 pre P'(z, w, v_2)
20 post Q'(z, w, v)
21 end_process;
22 process Sample_Process_3 (v: T_{o_3})u : T_{o_4}
23 transaction
24     Trans_1;
25 ext
26     wr v_3 : Te_3
27 pre P"(v, u, v_3)
28 post Q"(v, u, ~v_3, v_3)
29 end_process;
30 process Sample_Compensating_Process (v:
   T_{o_3})x : T_{i_1}, y : T_{i_2}, z : T_{o_1}, w : T_{o_2}
31 compensating transaction Trans_2;
32 ext
33     wr v_2 : Te_2
34 pre P"(v, v_2)
35 post Q"(v, x, y, z, w, ~v_2, v_2)
36 end_process;
37 end_module;
```

show its using scenario and effect. By introducing this, a more clear and deep understanding of extended SOFL will be able to build.

Our case is to model a workflow of purchasing commodity. It is quite common in web-based applications. The approximate processes are listed as follows.

1. Login to our system. In order to keep pace with popular SSO technology (stands for Single Sign On), user need to send his user name and password to a

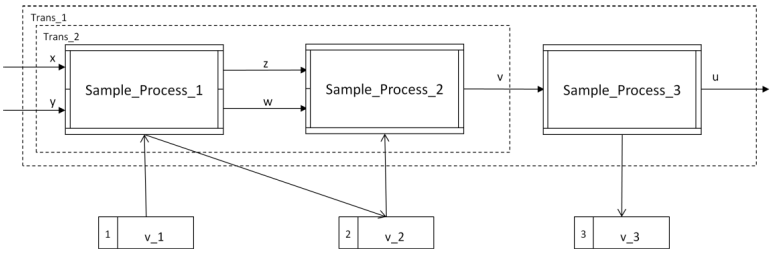

Fig. 3. CDFD of Module with Transaction

user authentication center. If authentication succeeds, UC provides our system with the user's ID along with some necessary user-related information. In general case, UC exposes a web service API to all its downstream system. Therefore a remote process of invoking that service is needed.

2. Online Payment. We have introduced a third-party online payment system to our workflow. Such systems includes PayPal or e-bank. It is also a remote process. After succeeding in payment, it returns flag indicating the successful message.

3. Increase Possessions. Add the number of the user's purchased commodity by 1 in user's profile. For data consistency, procedure No.2 and 3 must be "all-or-nothing", thus they must be in an transaction.

We use three processes to model this workflow, corresponding to three procedures mentioned above. Among these, login process and online-payment process are remote processes while the rest is local. Besides, we should introduce two local data stores saving user's authentication data and data about user's amount of possessions. The last key point is that the second and third process need to be included in a transaction to avoid situations such as failing in payment but succeed in adding possessions. The detailed information of these three processes are listed as follows.

1. Login. This is a remote process. It takes user's user name and password as input, and judge whether he is a valid user of our system. If true, it gives out detailed information of this user, otherwise an error message acts as its output. Dealing with this error message may need another process (maybe named '$Display_Message$'), but it is not the key point of our sample, thus it is omitted. '$Login$' also need to read information from data store named '$account\_info$', which saves user's authentication information.

2. Online Payment. This is also a remote process. It takes user's information as input and returns a flag indicating whether the payment operation is successful along with his information for next process. Actually it needs a **wr** operation to user's data about his balance, but it should not be displayed in our system because it is transparent to our workflow.

3. Increase Possessions. This is a normal process. It takes the current user as input and gives out a flag showing whether it succeeds. It needs to search the user's possession information by user's information and overwrite it, thus a **wr** operation to data store named '*possession_info*'.

After analyzing requirements and making a rough design about the system, we try to give out its detailed SOFL Specification of this workflow. Just as what we have discussed above, this specification contains remote processes and transactions. The module is named '*Purchae*'. It is shown in snippet 4.

We have omitted the process '*Display_Message*' which deals with the error message '*Login*' gives out. Moreover, as an example, we have also simplify the workflow. For instance, we assume there is only one type of commodity. In practice, information about commodities and their prices, stocks can always be very large. To transplant it into real use, we may need to add a process dealing with what commodities the user want to buy and their total prices between '*Login*' and '*Online_Payment*'. Also, logics dealing mistakes are also omitted such as when user's balance is not enough to afford or commodity is out of stock.

In this example, remote process and transaction is involved. We need to note some key points.

1. Post-conditions of remote processes are added by user, thus we cannot find the detailed implementation of online payment in post-condition of '*Online_Payment*'. We focus on their input and output variables to ensure the reasonability and correctness of the whole workflow.

2. We have not defined the compensating transaction for '*Purchase_Trans*', thus taking snapshot is used by default.

3. Data store '*account_info*' belongs to our system, thus it appears in CDFD, But user's data about his balance is transparent to our workflow, so it should not appear in CDFD.

4. If we try to use account information from other systems, '*account_info*' can be remote data store. This requirement is also common if several systems try to work together and share their users. The CDFD of this workflow is shown is figure 4.

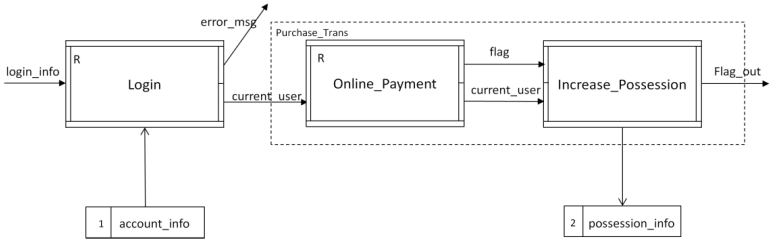

Fig. 4. CDFD of Purchase

Algorithm 4. SOFL Specification of Purchase

```
module Purchase;
type
  Login_Info = composed of
    user_name: string
    password: string
  end;
  Account_Info = composed of
    user_id: nat
    email: string
  end;
  LoginAccountFile = map Login_Info to Account_Info;
  PossessionAccountFile = map Account_Info to nat0;
var
  ext #account_info: LoginAccountFile;
  ext #possession_info: PossessionAccountFile;
inv
  forall[x: Account_Info] | not exists [y: Account_Info] | x.user_id = y.user_id;
transaction Purchase_Trans;
behav Purchase_CDFD;
remote process Login(login_info: Login_Info) current_user: Account_Info |
error_msg: string
ext rd account_info
post if login_info inset dom(account_info)
  then current_user = account_info(login_info)
  else error_msg = "Your password or user name is incorrect."
end_process;
remote process Online_Payment(current_user: Account_Info)current_user_out:
Account_Info, flag: bool
transaction Purchase_Trans;
post current_user_out = current_user
   and flag = true
end_process;
process Increase_Possession(current_user: Account_Info, flag: bool)flag_out:
bool
transaction Purchase_Trans;
ext wr possession_info
pre flag = true
post possession_info = override(~possession_info, map:current_user − >
~possession_info(current_user) + 1)
  and flag_out = true
end_process;
end_module;
```

5 Related Work

Workflow technology has attracted attentions of researches for a period of times. In workflow researching field, a representative language is BPEL[13]. Because of its ambiguities and lack of formal semantics, many researches have been performed to formalize BPEL by introducing process algebgra, Petri nets, automata, etc.

Process algebras is also used to form workflow. It contains ACP (stands for Algebra of Communicating Processes), CSP (stands for Communication Sequential Processes), CCS (Calculus of Communicating Systems), etc[14]. Researches on modeling business logic by process algebra is a popular topic these years. Salaun has presented a method for verifying business processes based on process algebras which mainly focus on their interactions[15]. The shortage of using process algebra to model business process mainly lies in its lack of support on dynamic process instantiation and correlation set. Process algebra also does not support dynamic structure alteration, which is important in business aspect.

Petri net is a strict and mathematical describing language. It can also used to verify workflow in an dynamic way. Using Petri net to model a business is an ideal and reliable method, especially after the introducing of high-order Petri net. Many researchers have tried their way to translate BPEL to Petri net[16][17][18]. But Petri net is based on graphical unit, thus its complexity boosts when modeling a large scaled and practical workflow. Moreover, data types in Petri net is also limited. Workflow involving rich data types is difficult to be described by Petri net.

According to the definition, automata is a public and base model of formal specification for systems which contains a set of stats, actions and transitions between states[19]. It is convenient to describe workflow because corresponding definitions can also be found in workflow. Diaz has researched a set of methods converting business processes writtin in BPEL-WSCDL to timed automata[20]. Fu has developed a tool which translating BPEL to guarded automata[21]. But limited by the feature of automata, it is also not suitable for describing large-scaled system because of the complicated structures and loss of accuracy.

Comparing to these relatively mature researches, SOFL starts later. The initial development of SOFL was made at the University of Manchester in the United Kingdom in 1989[22]. After that, SOFL had developed gradually with contribution of many researchers. Shaoying Liu has formalized its grammar and introduced to people in his papers and books.[5] A remarkable advantage of SOFL against other software formal language is its support of automatic verifying and validating. Several tools have also been developed to finish these[23][24]. Although its researches have not attract much people yet, its potential power and solid foundation is still convinced that it will keep going on.

6 Conclusion

As the rapid growth of software, both developers and users keeps changing. Because the number of software users boost, software need to face greater challenges

not only on its functional requirements, but also its performance, availability, quality, etc. Being different from classic testing method, modeling software workflow using formal language is another attractive way to ensure software quality. Besides, it is more convincible to people because its mathematical base.

In this paper, we have performed several extensions on grammar of SOFL in order to make it able to keep pace with the developing software world. These extension points are listed as follows.

1. Extensions on remote elements. This can further be divided into remote processes and remote data stores. We use remote processes to model invoking remote services in workflow, which is popular in SOA. Remote data stores are used to model invoking remote storage services in workflow. It is similar to remote processes. Remote elements are different from normal elements in several points. Its grammar on SOFL Specification mainly highlights in keyword **remote**.

2. Extensions on transactions. Transactions are important to software product because of its "all-or-nothing" feature. After extension, we are able to declare transaction name in module declaring section and add transaction declaring to process definition section to indicate which transaction(s) it belongs to. Transactions can be nested. One process can be included by multiple transactions. Its compensating transaction can be written by user, or using taking snapshots by default. All these two extra declaring section starts with keyword **transaction**.

We have also given out an simple example dealing with purchasing commodities to show the using scenarios and methods of our extensions. The changes we made to SOFL is not very large, and it does not conflict with normal SOFL grammar either because we use new keywords. These extensions can also be illustrated in CDFD by introducing different mark to distinguish.

SOFL is a new comer comparing with some elder and mature formal language members such as automata, Petri net and process algebra. But its potential power can not be regardless. As more researches have been done in SOFL, it is believed that SOFL will attracts increasingly more people including both researchers and users, thus ensure software quality in a more convincible way.

References

1. Marciniak, J.J.: Encyclopedia of Software Engineering, 2nd edn. Wiley Publications (1994)
2. Cai, L., Yang, G.: Software Quality Assurance Testing and Evaluating. Tsing Hua University Publications (2007)
3. Aked, M.: RUP in brief. In: Risk Reduction with the RUP Phase Plan, pp. 1–10. IBM (November 2003)
4. Pressman, R.S.: Software Engineering, a Practitioner's Approach. McGraw-Hill Science/Engineering/Math. (2009)
5. Liu, S.: Formal Engineering for Industrial Software Development. Springer (2008)
6. Liu, S.: 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)

7. Liu, S.: An automated rigorous review method for verifying and validating formal specifications. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 15–19. Springer, Heidelberg (2004)
8. Sun, C., Zhang, X., Zheng, L.: The research of the component-based software engineering. In: Sixth International Conference on Information Technology: New Generations, ITNG 2009, pp. 1590–1591 (2009)
9. Bell, M.: Introduction to Service-Oriented Modeling. Wiley and Sons (2008)
10. Raicu, I., Lu, S., Foster, I., Zhao, Y.: Cloud computing and grid computing 360-degree compared. In: Grid Computing Environments Workshop, GCE, pp. 1–10 (August 2008)
11. Benslimane, D., Dustdar, S., Sheth, A.: Services mashups: The new generation of web applications
12. Korth, H.F., Silberschatz, A.: Database System Concepts, 4th edn. McGraw-Hill Education (2006)
13. Morimoto, S.: A survey of formal verification for business process modeling. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part II. LNCS, vol. 5102, pp. 514–522. Springer, Heidelberg (2008)
14. Sipei, L., Jin, W., Lei, W., Park, S.: Description logic rule, matching process algebra based OWL-S modeling, and composition
15. Schaerf, M., Salaun, G., Bordeaux, L.: Describing and reasoning on web services using process algebra. In: Proceedings of the IEEE International Conference on Web Services, pp. 43–50 (2004)
16. Verbeek: Analyzing bpel processes using petri nets
17. Van der Aalst: Verification of workflow nets
18. Dumas, M., Van der Aalst, Verbeek, H.M.W.: An approach based on bpel and petri nets (extended version)
19. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley (2006)
20. Diaz, G., Pardo, J.-J., Cambronero, M.-E., Valero, V., Cuartero, F.: Automatic translation of WS-CDL choreographies to timed automata. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670, pp. 230–242. Springer, Heidelberg (2005)
21. Fu, X., Bultan, T., Su, J.: Analysis of interacting bpel web services. In: Proc. of 13th International Conference on the World Wide Web, pp. 621–630 (2004)
22. Sun, Y., Liu, S.: Structured methodology+object-oriented methodology+formal methods: methodology of sofl
23. Miyamoto, K., Liu, S., Fukuzaki, T.: A gui and testing tool for sofl
24. Wang, Y., Zheng, Q., Chen, H.: Soflipse: Tool for automatic modelling and reviewing sofl workflows. International Journal of Computing Technology and Information Security 1, 88–98 (2011)

Author Index

Chen, Haopeng 133

Duan, Zhenhua 71, 87

Kitagawa, Tetsuo 104
Kountanis, Dionysios 116

Li, Cencen 1
Li, Mo 1, 44
Ling, Chung-Ling 116
Liu, Shaoying 1, 12, 26, 44, 56, 71

Miao, Weikai 26

Nagoya, Fumiko 104
Nakajima, Shin 1

Shen, Wuwei 116

Tian, Cong 71, 87

Wang, Xi 56
Wang, Yisheng 133

Yang, Mengfei 87
Yu, Yan 87

Zhang, Weihang 12