Shaoying Liu
Zhenhua Duan (Eds.)

LNCS 8979

Structured Object-Oriented Formal Language and Method

4th International Workshop, SOFL+MSVL 2014
Luxembourg, Luxembourg, November 6, 2014
Revised Selected Papers

Springer

Lecture Notes in Computer Science 8979

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

More information about this series at http://www.springer.com/series/7407

Shaoying Liu · Zhenhua Duan (Eds.)

Structured Object-Oriented Formal Language and Method

4th International Workshop, SOFL+MSVL 2014
Luxembourg, Luxembourg, November 6, 2014
Revised Selected Papers

Springer

Editors

Shaoying Liu
Hosei University
Tokyo
Japan

Zhenhua Duan
Xidian University
Xi'an
China

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-17403-7 ISBN 978-3-319-17404-4 (eBook)
DOI 10.1007/978-3-319-17404-4

Library of Congress Control Number: 2015935418

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media
(www.springer.com)

Preface

There is a growing interest in building more effective technologies for future software engineering by properly integrating formal techniques into conventional software engineering process. The development of the Structured Object-Oriented Formal Language (SOFL) over the past two decades has shown some possibilities of achieving effective integrations to build practical formal techniques and tool support for requirements analysis, specification, design, inspection, review, and testing of software systems. SOFL integrates Data Flow Diagram, Petri Nets, and VDM-SL to offer a graphical 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. Meanwhile, the Modeling, Simulation, and Verification Language (MSVL) is a parallel programming language developed over the past decade. Its supporting tool MSV has been developed to enable us to model, simulate, and verify a system formally. The two languages complement each other.

Following the success of the previous SOFL workshops, the 4th international workshop on SOFL+MSVL (SOFL+MSVL 2014) was jointly organized in Luxembourg by Shaoying Liu research group at Hosei University, Japan, and Zhenhua Duan research group at Xidian University, China, with the aim of bringing industrial, academic, and government experts and practitioners of SOFL or MSVL to communicate and to exchange ideas. The workshop attracted 20 submissions on specification-based testing, specification inspection, model checking, specification animation, formal methods education, formal verification, formal semantics, and formal analysis. Each submission was rigorously reviewed by two or more PC members on the basis of technical quality, relevance, significance, and clarity, and 12 papers were accepted for publication in the workshop proceedings. The acceptance rate is 60 %.

We would like to thank the organizer of ICFEM 2014 for supporting the organization of the workshop, all of the Program Committee members for their great efforts and cooperation in reviewing and selecting papers, and our postgraduate students for their various helps. We would also like to thank all of the participants for attending the presentation sessions and actively joining in discussions at the workshop. Finally, our gratitude goes to the editors Alfred Hofmann and Christine Reiss of Springer for their continuous support for the publication of the workshop proceedings.

November 2014

Shaoying Liu
Zhenhua Duan

Organization

Program Committee

Shaoying Liu (Co-chair)	Hosei University, Japan
Zhenhua Duan (Co-chair)	Xidian University, China
Michael Butler	University of Southampton, UK
Yuting Chen	Shanghai Jiaotong University, China
Stefan Gruner	University of Pretoria, South Africa
Mo Li	Hosei University, Japan
Xiaohong Li	Tianjin University, China
Peter G. Larsen	Aarhus University, Denmark
Abdul Rahman Mat	Universiti Malaysia Sarawak, Malaysia
Huaikou Miao	Shanghai University, China
Weikai Miao	East China Normal University, China
Fumiko Nagoya	Aoyama Gakuin University, Japan
Shin Nakajima	National Institute of Informatics (NII), Japan
Kazuhiro Ogata	JAIST, Japan
Shengchao Qin	University of Teesside, UK
Wuwei Shen	Western Michigan University, USA
Jing Sun	University of Auckland, New Zealand
Cong Tian	Xidian University, China
Xi Wang	Shanghai University, China
Xinfeng Shu	Xi'an University of Posts and Telecommunications, China
Xiaobing Wang	Xidian University, China
Jinyun Xue	Jiangxi Normal University, China
Fatiha Zaidi	Université Paris-Sud, France
Hong Zhu	Oxford Brookes University, UK

Contents

Testing and Inspection

Model Checking and Animation

Education and Verification

Semantics and Analysis

Testing and Inspection

An Implementation Framework for Optimizing Test Case Generation Using Model Checking

Longhui Chang[1,2(✉)], Huaikou Miao[1,2], and Gongzheng Lu[2]

[1] School of Computer Engineering and Science, Shanghai University,
Shanghai 200444, China
{changlh,hkmiao}@shu.edu.cn
[2] Shanghai Key Laboratory of Computer Software Testing and Evaluating,
Shanghai 201112, China
lugz@shu.edu.cn

Abstract. Model checking based automated software testing has gained a great popularity in the field of software test. However, during the process of test cases generation, the redundant trap properties lead to calling the model checker frequently and generating redundant test cases. This paper presents an implementation framework for optimizing test cases generation based on satisfiability solving. After a new test case is generated, the SAT solver is employed to determine whether the generated test case covers the rest trap properties. If the trap properties are covered by the generated test case, they will be removed from the trap properties set. The bound model checker is used as the test generation engine, which effectively limits the length of counterexamples and ensures covering the same test goals with shorter total length. Our approach can not only decrease the times of calling the model checker, but also help to realize the automatic optimization of model checking based test cases generation.

Keywords: Bound model checking · Satisfiability · Test case generation · Trap-properties reduction

1 Introduction

With the advantage of automatic test cases generation, the software testing method based on model checking gets more and more attention. Model checking based test cases generation can be simply described as [1]: after the system under test (SUT) is formally modeled, the trap properties set are generated according to the model and coverage criteria. And then the model checker is used to verify whether the model satisfies the certain trap properties, and it will generate counterexamples automatically in the case of unsatisfiability. Automation process that uses model checker to generate test cases for each trap property, however, can often create a larger test cases set. And the redundant trap properties set that call model checker constantly will cause the consuming of system resources. Therefore, people have been focusing on optimizing test cases generation based on model checking. This paper proposes a solution to reduce trap properties set, which are generated by DFS algorithm, by deleting the trap properties that are covered

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 3–16, 2015.
DOI: 10.1007/978-3-319-17404-4_1

by generated test case in the process of test cases generation. Meanwhile, in order to control the total length of test cases set, we choose the bound model checker, which keeps a border k to limit length of the path to search. Since NuSMV model checker is already used on a formal verification of real-world application, it is selected as test generation engine in our implementation framework. And the commands of NuSMV help to get dimacs file of the SUT, which is used to call the SAT solver to verify whether the trap properties are covered by generated test case. This paper introduces an implementation framework from UML modeling to test cases generation and a case study running through the part of implementation framework shows the feasibility of our method.

The remainder of this paper is organized as follows: Sect. 2 describes background knowledge of model checking and SAT theory. The implementation framework for test cases generation is introduced in Sect. 3. The implementation and experiment of the framework are presented in Sect. 4. Section 5 summarizes some related work, and we finally conclude the paper and highlight our future work.

2 The Preliminaries

In this section, we introduce the basic knowledge of optimizing test cases generation based on model checking. Model checking starts formalizing the model of the SUT, and then chooses the right coverage criterion to generate trap properties set. As the model and trap properties are inputted into model checker, counterexamples will be generated when the trap properties are discovered to be false. In the process of optimization, the SAT theory is used to verify whether the generated test cases cover the rest trap properties.

2.1 Linear Temporal Logic

The most widely used temporal specification languages in model checking are LTL (Linear Temporal Logic) and CTL (Computation Tree Logic). In this paper, we mostly use the temporal LTL [2] to describe trap properties. The definition of LTL formula is given below:

$$\emptyset ::= \text{true}\ |\text{false}|\ a \in AP|\neg\emptyset|\emptyset_1 \wedge \emptyset_2|\emptyset_1 \vee \emptyset_2|\emptyset_1 \cup \emptyset_2|X\emptyset|F\emptyset|G\emptyset$$

An LTL formula consists of Boolean operators, atomic propositions and temporal operators. The “AP” is atomic propositions. The operator “X” refers to the “next” operator, “U” stands for “until” operator. The “G” is the “always” operator, stating that a condition has to hold at all states of a path, and “F” is the “eventually” operator that requires a certain condition to eventually hold at some time in the future.

Trap properties express the items that make up a coverage criterion by claiming that these items cannot be reached or covered. For example, a trap property may claim that a certain state can never be reached or a transition can never be covered. And the resulting counterexample shows how the state or transition described by the trap property is reached or covered. This counterexample can be used as a test case.

2.2 SAT Theory and Bounded Model Checking

Given a proposition formula, determining whether there exists a variable assignment that makes the formula evaluate to true is called the Boolean Satisfiability Problem (SAT) [3]. SAT was the first problem proven to be NP-complete problem. However, the efficient SAT solver is often successful in answering this problem. Given an SAT instance, SAT algorithms can completely either find a satisfying variable assignment, or prove that no such solution exists. In most implementations, the search is based on DPLL [4] algorithm proposed by Davis, Logemann and Loveland, and the proposition formula is usually expressed in a product of sums form which is usually called Conjunctive Normal Form (CNF). A formula in CNF is a conjunction of one or more clauses, where each clause is a constraint formed as the disjunction of one or more literals. A literal, in turn, is a Boolean variable or its negation. A propositional formula in CNF has some nice properties that can help prune the search space and speed during the search process. To satisfy a CNF formula, each clause must be satisfied individually. If a variable assignment causes any clause in the formula to make all its literals evaluate false, then that current variable assignment or any extension of it will never satisfy the formula. A clause that has all its literals assigned to value 0 is called a conflicting clause and directly indicates to the solver that some of the currently assigned variables must be unassigned first before continuing the search for a satisfying assignment. The DPLL algorithm is a depth-first backtracking framework. At each step, the algorithm picks a variable v and assigns a value to v. The formula is simplified by removing the satisfied clauses and eliminating the false literals. If an empty clause results after simplification, the procedure backtracks and tries the other value for v until all variables are valued. If the clauses set gets the true value, the CNF is defined satisfiability. Otherwise, it is unsatisfiability. Our primary task is to prove the unsatisfiability of the instance.

The bounded model checking (BMC) constructs an SAT instance that is satisfiable if and only if the underlying state transition system can realize a finite sequence of state transitions that reaches certain states of interest. The main idea of BMC is to turn SUT or model to Kripke, and express the properties to be verified in LTL formulae, then set the upper bound of the border k, and combine model and LTL formulae into SAT instance. If the property is verified to be false, then we will get a counterexample. On the contrary, it explains that the model meets the properties after running the k steps. For the border k, which is the maximum bound to be checked, if no value is given for it, the BMC will find the shortest counterexample which violate the model.

3 Implementation Framework for Optimizing Test Cases Generation

The framework for automatic test cases generation by model checking is shown in Fig. 1. On the one hand, the right coverage criteria are chosen to generate trap properties set. On the other hand, the SUT is converted to input program of model checking. After putting properties and SMV program into SMV model checker, it can generate counterexamples to indicate violated trap properties and such counterexamples are suited as test cases. The main goal of optimization in the process of test cases generation is to minimize trap properties set to avoid repeated trap properties calling NuSMV, thus to avoid wasting resources and generating redundancy test cases.

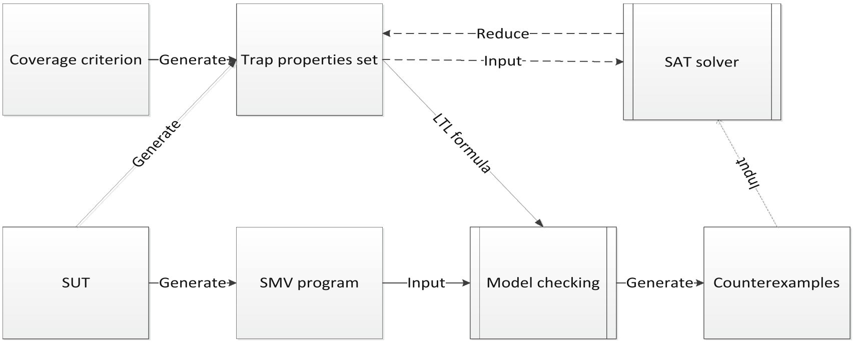


Fig. 1. The test case generation framework of model checking

3.1 UML Formal Modeling

The SUT can be modeled by the finite state machine (FSM), data flow diagram, and UML diagram. This paper uses statechart diagram of UML to describe the SUT. Figure 2 is the statechart diagram of an ATM. It mainly describes the user withdrawing money by ATM.

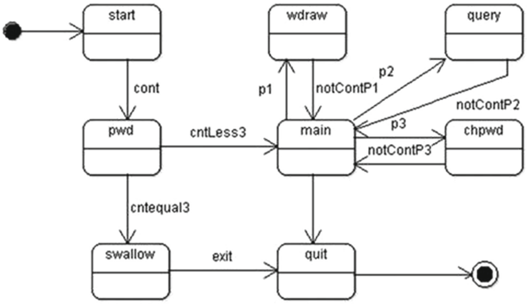


Fig. 2. ATM statechart diagram

ATM statechart diagram begins with a start action. After the user determines to continue (cont), the system will enter the password (pwd) state. If the password is correct, the system will transfer to the main interface automatically. Otherwise, return to the pwd state under the condition that the number of inputting password is less than three times. If the times is equal to three, the system will swallow card, and exit the system finally. In the main interface, user can input p1, p2, p3 to withdraw, query and modify password respectively, and can return to the main interface if the user does not continue other action. Figure 2 descripts the process of withdrawing from ATM, and embodies the transition logic and transition conditions between state nodes.

```
MODULE main
VAR
state: {s6,s4,s2,s3,s1,s7,s0,s5};
quit:boolean;
wdraw:boolean;
pwd:boolean;
… …
ASSIGN
init(state):=s0;
init(start):=TRUE;
init(quit):=FALSE;
init(wdraw):=FALSE;
… …
next(state):=case
    state=s4: {s1};
    state=s2: {s7};
    … …
    TRUE: state;
esac;

next(quit):=case
    next(state)=s6: TRUE;
    TRUE: FALSE;
esac;
… …
next(notContP1):=case
    next(state)=s5: TRUE;
    TRUE: FALSE;
esac;
… …
```

Fig. 3. The partial SMV program for ATM

3.2 From UML to SMV Procedure Model

The Unified Modeling Language (UML) is the standard visual language used for modeling the software behavior. So we choose the UML to describe the system model. But while we call the model checker, the UML model should be converted to SMV program. A SMV program is composed of a set of modules. The VAR module declares the states and local variables of the model. The transition relationships are captured in the ASSIGN module.

The process to generate SMV procedure consists of two steps. Firstly, the ArgoUML tool is used to convert UML diagram into XML document which is up to XMI (XML-based Metadata Interchange) standard descripting the UML diagram completely. The XMI not only unifies exchange of a large number of information, but also regulates XML document transformed from UML model and helps to achieve the mapping from UML diagrams to XML.

Secondly, the states set, transitions set and triggering events are extracted from statechart diagram to structure the SMV program by parsing the XMI document. In the VAR module of SMV program, all states are represented with enumeration type and initialized. Transitions are converted to a series of next statements by defining with case statements. We only give the part of SMV program for ATM example for paper limitation, as shown in Fig. 3.

3.3 Encoding Test Coverage Criterion into Trap Properties

A model checker will be called to find counterexamples by formulating a coverage criterion as a verification condition for the model checker [6]. The coverage criterion available for statechart diagram includes state coverage, transition coverage, and transition composition coverage. In this paper, we choose the state coverage criterion as the example, which can be considered as reachability property. Counterexamples generated by reachability properties satisfy the state coverage if all states of the model are covered at least once by them. Reachability properties defined in LTL formulae begin with the keyword LTLSPEC in SMV model. Following the LTLSPEC, the reachability property is defined by "G" operator.

Table 1. Reachability properties for ATM

No.	State	Trap property
TP1	chpwd	G!chpwd
TP2	query	G!query
TP3	quit	G!quit
TP4	main	G!main
TP5	wdraw	G!wdraw
TP6	swallow	G!swallow
TP7	start	G!start
TP8	pwd	G!pwd

For example, to reach the *pwd* state in Fig. 2, the following trap property is used: $\emptyset :: = \mathrm{G!pwd}$. The negation means that no transitions are existed to reach state *pwd*. Then we will get the counterexample which does not satisfy the trap property but is the test sequence reach the state *pwd*. In the process of generating trap properties, we employ the depth first search algorithm to traverse all the states, which, on certain extent, ensures that the trap properties with higher priority has higher coverage rate than those lower priority at the default sort. All trap reachability properties of ATM of Fig. 2 are listed in Table 1.

3.4 Generate Test Cases and *Dimacs* File

The BMC takes a model and the properties as input, a counterexample will be generated when the property violates the model. Here we encapsulate the NuSMV commands [7] of generating counterexamples by BMC into a .bat file, shown in Fig. 4. The model checker reads the SMV program and sets up *bmc*. We can get the state sequence with the command *show_traces* and the total execution time with *time* command.

```
@echo off
color 0a
(echo read_model -i D:\ ATM.smv;
echo flatten_hierarchy;
echo encode_variables;
echo build_boolean_model;
echo bmc_setup;
echo gen_ltlspec_bmc;
echo check_ltlspec_bmc_inc;
echo show_traces -a -p 4 -o
D:\NuSMV_Program\result\ATM_traces.xml;
echo time;
echo quit;
)|NuSMV -int
pause
```

Fig. 4. .bat for generating test cases

Conceptually a test case is a single trace through the state machine. We can express the test case as a constrained finite state machine, or CFSM [6], by adding special variables, states, which control the machine. The CFSM of the counterexample (*start, cont, pwd, cntqual3, swallow, exit, quit*) generated by TP1 is shown in Fig. 5. Expressing the generated test case as a CFSM helps us to restore the test case to a model.

```
VAR
State: s0, s1, s2……;
ASSIGN
init (state) := s0;
init (start) :=TRUE;
init (pwd) :=FALSE;
init (swallow) :=FALSE;
init (cont) :=FALSE;
……
next (start) := case
  next (state) = s0 : TRUE;
  TRUE : FALSE;
esac;
next (pwd) := case
  next (state) = s1 : TRUE;
  TRUE : FALSE;
esac;
next(cntqual3):=case
  next(state)=s1: TRUE;
TRUE: FALSE;
esac;
……
next (state) := case
state=s0: {s1};
state=s1: {s2};
……
esac;
```

Fig. 5. The partial CFSM program for test case generated by TP1

As we convert the generated test case to the CFSM, we use the NuSMV commands, shown as the following, to generate Dimacs file for SAT solver, to verify the consistency of the model and rest trap properties.

```
read_model -i model-file
flatten_hierarchy
encode_variables
build_boolean_model
write_boolean_model
bmc_setup
gen_ltlspec_bmc  -o ***.dacmcs
```

The input, model-file, refers to the CFSM. And the output is a *dimacs* file which contains of CNF formulae set. Then the SAT solver is called to search a variable assignment for the CNF formulae. If the result is false, it indicates that generated test case has covered the current property. Then the property should be deleted before it calls NuSMV.

3.5 Reduction of Trap Properties Set

In the simple approach that test cases are generated by calling the model checker for each trap property, which results in a test case for a feasible trap property. It means that the same or a similar test case might be created several times, or that a test case might be a prefix of another test case because a test case may contain several trap properties violation. If the number of trap properties is too large, more time will be cost to create redundant test cases.

```
Input: TP[]//the DFS trap properties set; M//system model
Output: TC[]//test case set

while TP[] is not empty do
   TC[i] = Modelchecking(M, TP[i]);//call model checker
   if fail then
      Cmodel=CFSM(TC[i]);
      for the rest trap property in TP[]
          d=GenerateDimacs(Cmodel, TP[j]);
          result = SAT(d);//solve satisfiability
          if result = unsatisfiable;
             Then TP[]=TP[]\TP[j];//delete trap property
          end if
          else j++;
          end else
      end for
   end if
end while
Output TC[];
```

Fig. 6. The algorithm for reducing trap propertie

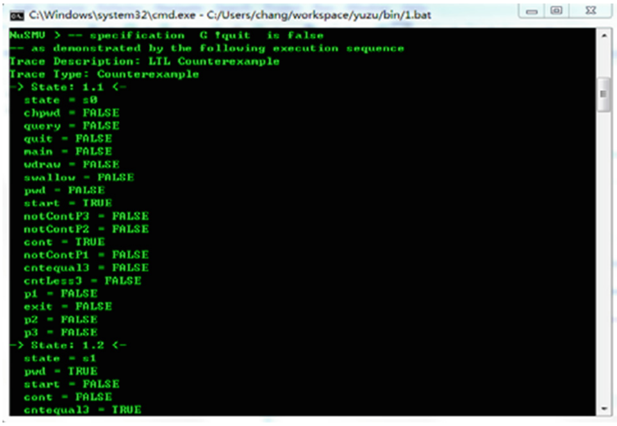
To avoid the above phenomenon, the test cases generation should be monitored. In this paper, we consider the SAT instance of the generated test case and the rest trap properties to verify whether the generated test case covers the rest trap properties. As we convert the consistency verification of the rest trap properties and generated test case into the CNF formulae, each state will be assigned a variable automatically. And the satisfying assignment that is found can be decoded into a state sequence which reaches states of interest [3]. If the result of SAT solver is false, which means there exists a state sequence in the model violates the trap property according to the SAT theory at Sect. 2.2, the trap property is duplicated with the generated test case. Then the trap property can be removed from the trap properties set. For example, if the counterexample generated for the first trap property is $tc_1 = s_0 s_1 s_2 s_3 s_4$, another counterexample corresponding with the third trap property is $tc_3 = s_0 s_1 s_2$. Then the *tc*$_3$ will not be generated for that *tc*$_3$ is contained by *tc*$_1$ as its prefix. As the trap properties are generated by DFS algorithm, the trap properties with higher priority largely cover those with lower priority, which on certain extent ensures that as many redundant trap properties as possible are deleted and the calling of the model checker is avoided. The algorithm for optimizing the trap properties set which based on satisfiability using SAT solving is shown in Fig. 6.

We input the ATM model and trap properties set TP[] that contains all trap properties ordered by DFS according to state coverage criterion, the output is a reduced test cases set TC[]. Firstly, the trap properties should be described in LTL. While the TP[] is not empty, model checker verifies TP[i]. Once a new counterexample is generated as the model violates the trap property, the SAT solver is called for checking whether the rest trap properties are covered by the generated counterexample. If the result of SAT solver is unsatisfiability, the trap property will be removed from TP[]. Otherwise, repeat the process until the TP[] is empty. Once the trap properties that are covered by generated counterexample are removed, it will not call model checker to generate repeated test cases.

4 Implementation and Experiment

We implemented the techniques described above in our test cases generation tool. We chose the Eclipse as the development platform because it supports many plug-ins, such as ArgoEclipse, and it has ability to add customize modules. After the formal modeling, the UMLEclipse we added translates UML diagram defined by the user into an XMI file, as described in the Sect. 3.2. From there, we are able to construct the input file of model checker. The SMV program generation module, using the API provided by Dom4j, is called to parse the XMI file by extracting the states set, transitions set and triggering evens set and generates the SMV program. The trap properties based on coverage criteria that chosen by the user can be generated according to the extraction result of parsing the XMI file, based on the construction given in Sect. 3.3. The parsing process is implemented by calling the common JUnit test execution engine to run the JAVA files. We encapsulate the NuSMV commands into a *.bat* file and it is invoked to call model checker to generate counterexample, which corresponds with the certain trap property. Then we can convert this new counterexample to CFSM by adding the states and transitions according to the Sect. 3.4. Using another *.bat* file, encapsulated the commands of SAT solving, we verify whether the rest trap properties are redundancy. The redundancy trap properties whose results are false are removed from the trap properties set according to the optimization algorithm described in the Sect. 3.5. We will maintain a loop to keep generating counterexample one by one. So once a new counterexample is generated, it will be used to check the rest trap properties and we will reduce the trap properties set. This paper only chooses the state coverage as the example to explain the feasibility of our tool. In fact, there are transition coverage and composition coverage properties available in our tool to generate different counterexamples to meet the requirements of user.

Using our test cases generation tool we experimented with the ATM example described in Fig. 2. And the Fig. 3 and Table 1 show the SMV model and trap properties respectively. The result of running the model checker for verifying the first trap property is shown in Fig. 7.

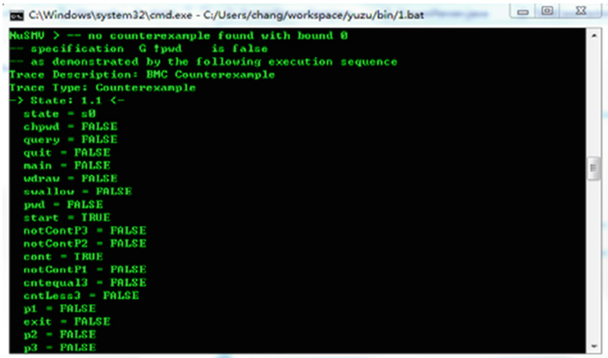


C:\Windows\system32\cmd.exe - C:/Users/chang/workspace/yuzu/bin/1.bat

```
NuSMV > -- specification  G !quit  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = s0
  chpwd = FALSE
  query = FALSE
  quit = FALSE
  main = FALSE
  wdraw = FALSE
  swallow = FALSE
  pwd = FALSE
  start = TRUE
  notContP3 = FALSE
  notContP2 = FALSE
  cont = TRUE
  notContP1 = FALSE
  cntequal3 = FALSE
  cntLess3 = FALSE
  p1 = FALSE
  exit = FALSE
  p2 = FALSE
  p3 = FALSE
-> State: 1.2 <-
  state = s1
  pwd = TRUE
  start = FALSE
  cont = FALSE
  cntequal3 = TRUE
```

Fig. 7. The result of running model checker for trap property TP1

As the trap property is false, the trace will be generated. This counterexample in Fig. 7 denotes *pwd* is true in next state s1, which violates the given property, i.e., there exists a path from start to *pwd*. The same to the next state. Then the counterexample constructed from SMV only shows the variables that value have changed in next state, such as *pwd* changes from FALSE to TRUE. After getting the counterexample, we can build the CFSM for the first counterexample shown in Fig. 4. Another .bat file is ran to call the Zchaff solver and the computational result is shown in Fig. 8. Here, we just focus on that the trap property is true or false, and do not consider the state trace. As the trap property is false, we will remove it according to our optimization theory. The value of bound, *k*, explains that it takes *k* steps to reach the target state.



C:\Windows\system32\cmd.exe - C:/Users/chang/workspace/yuzu/bin/1.bat

```
NuSMV > -- no counterexample found with bound 0
-- specification  G !pwd    is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = s0
  chpwd = FALSE
  query = FALSE
  quit = FALSE
  main = FALSE
  wdraw = FALSE
  swallow = FALSE
  pwd = FALSE
  start = TRUE
  notContP3 = FALSE
  notContP2 = FALSE
  cont = TRUE
  notContP1 = FALSE
  cntequal3 = FALSE
  cntLess3 = FALSE
  p1 = FALSE
  exit = FALSE
  p2 = FALSE
  p3 = FALSE
```

Fig. 8. The result of running SAT solver

After the tool repeats above operation, the final result of optimizing test cases generation is described in Table 2. The third column lists the trap properties that cover current trap property. For example, TP4 is covered by the test case generated by the trap property TP1, and TP6 is covered by TP3. If the trap property is covered by the generated test case, this trap property will be removed. So the TP4, TP6, TP7, TP8 will be deleted before they call the model checker. Without using our optimization, it will be 8 test cases and calls model checker 8 times. Here, we just generate 4 test cases and the number of times of calling the model checker is reduced. By analyzing the results, the optimization method reduces the test cases set effectively.

Table 2. The result of opimization based on SAT

Trap Property	Test case	Be covered TP	Delete
TP1: G!chpwd	<start, pwd, main, chpwd>		N
TP2: G!query	<start, pwd, main, query>		N
TP3: G!quit	<start, pwd, swallow, quit>		N
TP4: G!main		TP1	Y
TP5: G!wdraw	<start, pwd, main, wdraw>		N
TP6: G!swallow		TP3	Y
TP7: G!start		TP1	Y
TP8: G!pwd		TP1	Y

In bound model checking, the border *k* helps us to limit the length of counterexamples, and the total length of counterexamples is shorter than that produced by normal checker. Table 3 shows the total length comparison of our optimized method with non-optimized for the state and transition coverage criteria. The *87* expresses the total length of counterexamples generated by normal model checker without optimization, and the *28* expresses the total length of counterexamples generated by BMC with our optimized method, which shrinks by 60 %. The result shows that the total length of counterexamples generated by BMC is shorter than those produced by normal model checker, even under non-optimized method case, which reflects the advantage of BMC. Based on this analysis, our optimized method reducing the test set can ensure that the less test cases with shorter total length cover the same test goals.

Table 3. The result of optimization method based on SAT

Total length of optimized/non-optimized	MC(Model Checker)	BMC
State coverage	43/87	28/42
Transition coverage	47/112	34/68

5 Related Work

Testing based on model checking has obtained lots of achievements, but a few for bound model check. Such as, Moonzoo Kim and Yunho Kim [8] applied SAT-based bounded analysis to embedded software verification. Kelson Gent and Michael S. Hsiao [9] used BMC to generate functional test at the register transfer level. These method all utilized the bounded characteristic of BMC.

At the same time, many approaches dedicating to optimize the generation process based on model checking have been proposed. For example, Fraser et al. [10] used the LTL rewriting rules in the test generation process to eliminate the trap properties that have been covered, and at the same time produced less redundant test suite. Hamon [11] extended the existing test cases iteratively to meet the other trap properties in SAL integration. But this method is only suitable for those open programming interface model checking. Godefroid et al. [12] put forward the optimization of CTL and LTL semantics. In this paper, we implement a framework for optimizing test cases generation using model checker based on SAT. Our approach for optimizing the test cases generation process is effective and realizable.

6 Conclusion

Test automation based on model checking has become a hot spot in software testing. Although many optimization approaches has been presented, its commonality and implementability is not strong. The novel method proposed by this paper will reduce the test suite based on SAT theory by deleting the trap properties covered by generated test case. We give a detailed implementation steps for reducing the trap properties from UML formal modeling to test cases generation. The result of case study shows that our method not only decreases the times of calling model checker, but also shrinks the total length of the test cases and reduces test cases set to avoid the possibility of test cases space explosion. In the future work, we will pay more attention to the influence of boundary value *k*, which may affect the length of test cases set and the coverage rate of test goals. And at the same time, we will devote ourselves to implement the complete automation for the process of optimizing test cases generation.

Acknowledgments. This work is supported by National Natural Science Foundation of China (NSFC) under grant No. 61170044 and 60970007, and Innovation Program of Shanghai Municipal Education Commission (No. 13YZ141).

References

1. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
2. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, 31 October–2 November, Providence, Rhode Island, USA, pp. 46–57. IEEE (1977)

3. Mahajan, Y.S., Fu, Z., Malik, S.: Zchaff2004: an efficient SAT solver. In: SAT (Selected Papers), pp. 360–375 (2004)
4. Müller-Olm, M., Schmidt, D.A., Steffen, B.: Model checking: a tutorial introduction. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 330–354.Springer, Heidelberg (1999)
5. Li, L., Miao, H., Chen, S.: Test generation for web application using model-checking. In: 2010 11th ACIS International Conference, software engineering artificial intelligence networking and parallel/distributed computing (SNPD), pp. 9–11 (2010)
6. Chvatal, V.: A Greedy heuristic for the set-covering problem. mathematics of operations research. Math. Oper. Res. **4**(3), 233–235 (1979)
7. Cavada, R., Cimatti, A., Jochim, C.A., Keighten, G., Olivetti, E., Pistore, M., Roveri, M., Tchaltsev, A.: NuSMV 2.5. User Manual.
8. Kim, M., Kim, Y., Kim, H.: A comparative study of software model checkers as unit testing tools: an industrial case study. IEEE Trans. Soft. Eng. **37**, 146–160 (2011)
9. Gent, K., Hsiao, M.S.: Functional test generation at the RTL using swarm intelligence and bounded model checking. In: 2013 22nd Asian Test Symposium, pp. 233–238 (2013)
10. Fraser, G., Wotawa, F.: Using LTL rewriting to improve the performance of model-checker based test-case generation [C]. In: Proceedings of the 3rd International Workshop on Advances in Model-Based Testing, pp. 64–74. ACM Press, New York (2007)
11. Hamon, G., de Moura, L., Rushby, J.: Generating efficient test sets with a model checker. In: Proceedings of the 2nd International Conference on Software Engineering and Formal Methods, pp. 261–270. IEEE Computer Society Press, Los Alamitos (2004)
12. Godefroid, P., Huth, M.: Model checking vs. generalized model checking: semantic minimizations for temporal logics. In: Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS 2005), Chicago, pp. 158–167. (2005)

Applying GA with Tabu list for Automatically Generating Test Cases Based on Formal Specification

Yuqin Zhou[1], Taku Sugihara[2], and Yuji Sato[1,2](✉)

[1] Graduate School of Computer and Information Sciences,
Hosei University, Tokyo, Japan
[2] Department of Computer and Information Sciences,
Hosei University, Tokyo, Japan
yuji@k.hosei.ac.jp

Abstract. How to generate adequate test cases based on a specification to cover all of the paths in its implementation is a challenge in software testing. This paper presents a new approach to selecting test cases for program testing. The essential idea is to generate a set of test cases based on a given operation specification in pre-post notation, and then apply the genetic algorithm to facilitate the generation of more effective test cases in terms of program path coverage. The principle of GA is discussed and an improvement of the GA through integration with Tabu list is presented. An experiment is conducted to study how the improved GA can be applied and to evaluate its effectiveness. The result shows that our proposed method is more efective than conventional methods and can cover all paths based on formal specification.

Keywords: Formal specification · Genetic algorithm · Tabu list

1 Introduction

Most of testing projects try to adopt automatic test case generation to improve productivity and coverage rate. An important step in automatic testing is to select appropriate test cases. An effective application of formal specification in practice is facilitating test case generation, and the automation of the program testing process [13]. The automatic specification-based testing (ASBT) is a potentially effective technique for software reliability and attractive to the software industry, which can reduce the cost and time and avoid many human errors at the testing process. More and more companies adopt the formal specification to the requirement analysis or system design, this makes the automatic specification-based testing become practical. ASBT has proposed several approaches that let the test cases be generated only based on specifications. The criteria for determining if test cases satisfy the test condition have been put forward, but it does not ensure that every path in the related program can be traversed. Now the problem is how to ensure the paths in the related program can be traversed completely.

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 17–31, 2015.
DOI: 10.1007/978-3-319-17404-4_2

To solve the above problem, "Vibration" method has been proposed [12]. It has improved path coverage rate dramatically, however it still can not ensure to cover all paths in the representative program. There is also a large body of research on specification-based testing. Many projects automated test cases generation from specifications, such as Z specification [1,2], UML statecharts [3,4], or ADL specifications [5,6]. The Korat which is one of the novel frameworks for automated testing of Java Programs, is based on Java predicates [7]. Given a predicate and a bound on the size of its inputs, Korat generates all inputs for which the predicate returns true. Marisa A.S'anchez has examined algebraic-specification based testing with particular attention to the approach reported by Gilles Bernot, Marise Claude Gaudel and Bruno Marre [8]. It did not only focus on the generation of test cases from the initial specification but also the additions and revisions during the development. D.Marinov and S. Khurshid presented a framework for automated testing of Java programs called TestEra [9]. TestEra uses the Alloy Analyzer (AA) [10] to automatically generate method inputs and check correctness of outputs, but it requires programmers to learn a specification language much different than Java. Cheon and Leavens have proposed an automatic translation of JML specifications into test oracles for JUnit [11]. But the programmers have to provide sets of possibilities for all method parameters, which add a burden of test cases generation for programmers.

In this paper, Genetic Algorithm (GA) has been used to address the problem by studying how to ensure the paths in the related program can be traversed completely. Roy P. Pargas, Mary J. Harrold, and Robert R. Peck [18] have already applied standard genetic algorithms to generate a test data for a specific path. On the other hand, we propose improved genetic algorithms to generate test cases for all paths in the related program. A large number of test cases are generated by GA, and the appropriate test cases have been chosen after genetic manipulation to cover all paths in the related program. In addition, we improve the algorithm by integrating the Tabu search method.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction to the formal specification and automatic test cases generation. In Sect. 3, we present how to apply GA to solve this issue and improve the GA using Tabu search method. In Sect. 4, we present the result of the experiments. In Sect. 5, we conclude the study and point out some problems in practice we need to solve in the future.

2 Formal Specification and Automatic Test Case Generation

Our work proposed in this paper is based on the automatic specification-based testing(ASBT) method by Liu in [13], it is therefore necessary to introduce briefly Liu's work for readability of this paper. Automatic specification-based testing (ASBT) is a potentially effective technique for software industry, by avoidance of many human errors during a testing process to save development cost and time significantly. In the formal specification, we let $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ represent the specification of an operation S, where S_{iv} is the set of all input variables whose values are not changed by operation S, S_{ov} is the set of all output variables whose values are produced or updated by the operation, and S_{pre} and S_{post} are the pre- and post-condition of S, respectively. Each formal specification $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$ can be transformed into an equivalent functional scenario form (FSF): $(\sim P_{pre} \wedge C_1 \wedge D_1) \vee (\sim P_{pre} \wedge C_2 \wedge D_2) \vee ... \vee (\sim P_{pre} \wedge C_n \wedge D_n)$. $\sim S_{pre} \wedge C_1 \wedge D_1$ is called a functional scenario.

In the functional scenario, the tilde mark $\sim$ in $\sim S_{pre}$ represents the initial values of the variables in pre-condition before operation; C_i is a guard condition which is derived from S_{post}, which includes only input or initial state variables; $\sim S_{pre} \wedge C_i$ is test condition. Each test condition can be transformed into an equivalent disjunctive normal form: $P_1 \vee P_2 \vee ... \vee P_m$, each P_i presents a conjunction of atomic predicates. Each P_j consists of Q_i, in the format like this:$Q_1 \wedge Q_2 \wedge ... \wedge Q_w$. To generate a test case based on formal specification is to generate a test case by satisfying the atomic predicates in the test condition. This process can be decomposed into two steps:

Step1: Generating a test case t to satisfy the Q_1

Step2: If t satisfies the all other atomic predicates, t is a test case; otherwise, go back to *Step1* until finding a test case that satisfies all the atomic predicates or to be stopped by human operation.

The process is illustrated in Fig. 1. Each note denotes a state that satisfies all the predicates expressions along the "path" from the starting state S0 to itself; and each edge denotes a predication. Therefore, the S1a represents a state that satisfies all predicates along the "path", from Q11 to Q1a. The dotted line in the graph represents omission of many intermediate "state transitions". All the states S1a, S2b, . . . ,Smv are called accepting states, each of which represents a state that satisfied all the predicates along the path from S0 to itself. For example, S1a denotes a state that satisfies the conjunction Q11∧Q12∧ . . . ∧ Q1a [12,13].

The essential idea of ASBT is to generate a test set that covers every scenario by satisfying its test condition at least once. Generally, this problem may be handled by generating more test cases, but what test cases should be generated

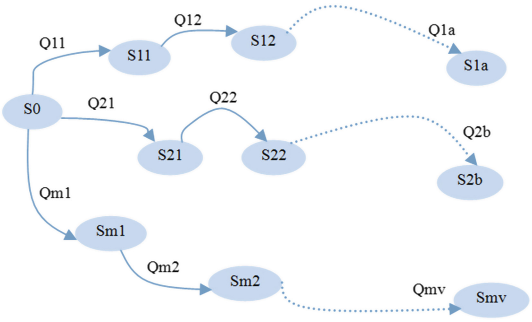


Fig. 1. The graph of the disjunctive normal form of a testing condition

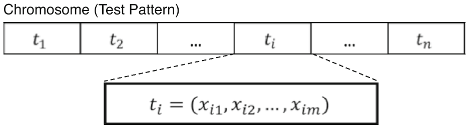
is still a big challenge. Liu has proposed the “Vibration method” which tries to change the “distance” between the variables to find more test cases to cover all corresponding program paths. For example, if there are two expressions E1 and E2, the “distance” (| E1-E2 |) between E1 and E2 will be changed as the values of variables in two expressions changed. A test case is created to satisfy the relation when the distance is small; another new test case will be created when the distance is greater. Repeating this process by increasing and decreasing the distance between E1 and E2 until the terminated decision is made. Although this method improved path coverage rate dramatically, it can not ensure that all paths will be covered since the distance is changed randomly at every time. In this paper, we introduce the GA to improve the performance of coverage rate in ASBT. And then we improve the GA using Tabu list, to guarantee a better performance of GA in automatic test cases generation. In next section, we will describe the procedure of how to apply GA to generate test cases based on formal specification [13].

3 Applying GA to Automatic Test Case Generation

In this section, we propose applying GA to automatic test case generation based on formal specification. By utilizing GA, the process of choosing test cases will be more effective. Initially, GA generates a population based on the specification, in which each individual is a set of test cases. The individual who has high path coverage rate will be selected to produce next generation. After the GA manipulation, the population in the next generation will be better and have a higher path coverage rate. The evolution will continue until all the paths in the program have been traversed.

3.1 Definition of Chromosome for Automatic Test Case Generation

The first step of GA is to randomly generate population of chromosomes [16]. Each member in the population is called individual. In our work, an individual is a test set consisting of one or more test cases. The test cases are generated based on the formal specification. If the chromosome is defined as a test case, it is difficult to judge whether the chromosome is good enough to be used to generate next generation. Because every chromosome will be in only two states:one is covered a path; the other is not cover a path. If the chromosome is defined a set of test cases, there will be a path coverage rate for every chromosome, and then we can judge which one is appropriate to be selected to generate the next generation. Figure 2 shows an example of a chromosome definition for a set of test cases. Every genetic locus in each chromosome corresponds to a test case, which satisfies the precondition. For example, the input variables are numeric type from 1 to 100, and then genetic locus will be defined as a number from 1 to 100 randomly. The length of the chromosomes (how many test cases in a chromosome) equals to the number of the paths. Thus, in the Fig. 2, t_i represents a test case, and n denotes the number of the paths. By generating a large scale of chromosomes in a population, we can generate a lot of test cases satisfied the precondition.


Chromosome (Test Pattern)
t_1
t_2
...
t_i
...
t_n
$t_i = (x_{i1}, x_{i2}, \dots, x_{im})$

t_i: Test case, x_{ij}: Input parameters,
n: Total number of the path, m: Total number of the input parameters

Fig. 2. Chromosome definition for test cases generation

Figure 3 shows an example of the initial population when we don't have any previous test patterns or knowledge to generate test patterns from corresponding paths mapIn this example, input parameters are generated randomly in a feasible region. Figure 4 shows an example of the initial population when we have some previous test patterns or knowledge to generate test patterns from corresponding paths map. In this example, we use these test patterns as a part of chromosomes and generate other input parameters randomly.

3.2 Evaluation Function

In a genetic operation, the evaluation function is a fitness function which is devised for each problem to be solved. Given a particular chromosome, the fitness function returns a single numerical "fitness", or "figure of merit", which is supposed to be proportional to the "utility" or "ability" of the individual which that chromosome represents. In this case, the fitness function is defined as the path coverage rate, when the chromosome covered more paths, its evaluation value will be higher. In this study, we assume that the number of all paths in the related program is known and define the fitness function in Eq. (1) below, which is normalized to values between 0 to 1.

$$f = \frac{k}{n} \tag{1}$$

In Eq. (1), f is the evaluation function, k is the number of the paths that chromosome has covered; n is the number of all paths in the corresponding programs. The individual who achieves an evaluation value f=1 means that the individual has covered all paths in the corresponding programs, and it is the optimal solution to the problems.

3.3 Genetic Manipulation

Selection. Selection is the stage in which individual generates for next generations from a population based on fitness values. There are many kinds of selection methods in GA, and we apply tournament selection [16] in this study. Tournament selection has a simple rule and can guarantee that the better one will be chosen and the worse one will be eliminated. Tournament selection involves randomly picking several individuals from the population and staging a tournament to determine which one is finally selected. It generates a random value between

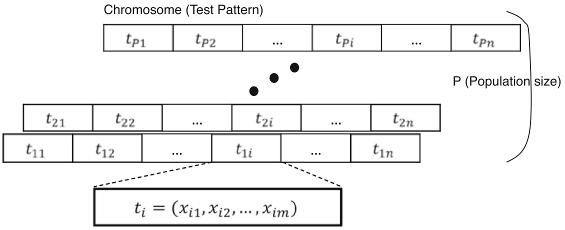


Fig. 3. An example of initial populations when we don't have any previous test patterns

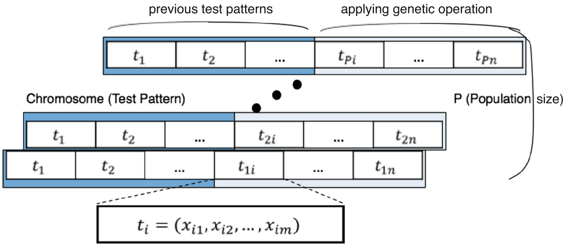


Fig. 4. An example of initial populations when we have some previous test patterns

zero and one and comparing it to a pre-determined selection probability. If the random value is less than or equal to the selection probability, the fitter candidates is selected; otherwise, it select other candidates again. The probability parameter provides a convenient mechanism for adjusting the selection pressure. The tournament size is 3, selecting the highest fitness one to be parent for next generation by comparing the three individuals. The individuals selected by tournament selection as parents to do crossover.

Crossover. Crossover is a genetic operator used to vary the programming of chromosomes from one generation to the next. In this paper we choose two-point crossover [16] since it can keep the stability of the individuals' value of fitness when the average fitness in the population is high enough. There is a variable t, which is generated randomly. When it is bigger than the crossover rate, the crossover will take two individuals, and cuts their chromosome strings at some randomly chosen position, to produce two "head" segments, and two "tail" segments. The two segments between the "head" and "tail" are then swapped over to produce two new full length chromosomes. The two offsprings each inherit some genes from each parent.

Mutation. Mutation is a genetic operator used to maintain genetic diversity. When a number of the input parameter is 1, we can apply a simple mutation. An example of a simple mutation is shown in Fig. 5. Mutation is applied to each child individually after selection and crossover. It randomly alters each gene with a small probability. If the variable t which is generated randomly is bigger than

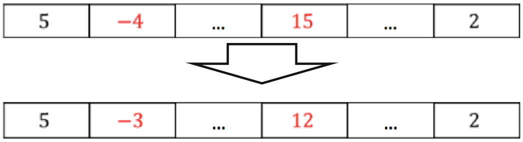


Fig. 5. An example of a simple mutation

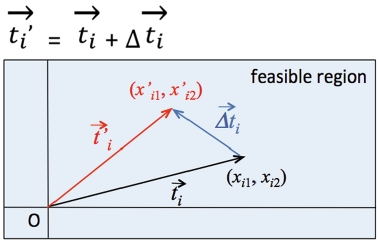


Fig. 6. An example of a different vector mutation for the two input parameters

mutation rate, the mutation manipulation will be executed. Mutation provides a small amount of random search, and helps ensure that no point in the search space has a zero probability of being examined. When a test case is consisted of multiple input parameters, a mutation is defined as a different vector mutation. Figure 6 shows an example for the two input parameters.

Repeat the above steps until the individuals whose fitness is 1 have been found or the termination condition has been met [14–16].

3.4 Proposal of GA with Tabu List for Covering Paths

We design test cases generation method based on the standard GA to find all paths in the corresponding program. Figure 6 shows an example of the paths in a program, in the form of binary tree. Gene has been designed as a test case,

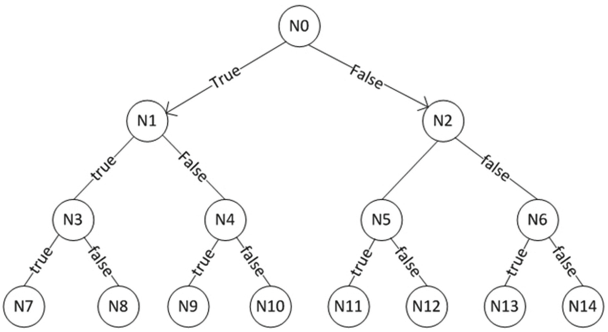
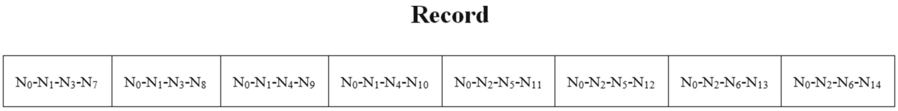


Fig. 7. A path map of a test program



Record

N_0-N_1-N_3-N_7	N_0-N_1-N_3-N_8	N_0-N_1-N_4-N_9	N_0-N_1-N_4-N_{10}	N_0-N_2-N_5-N_{11}	N_0-N_2-N_5-N_{12}	N_0-N_2-N_6-N_{13}	N_0-N_2-N_6-N_{14}

Fig. 8. An array records paths which have been found like a Tabu list

and the number of genes contained in each individual equals to the number of the paths. There are 8 paths in Fig. 7, each node is a predicate extracted from the formal specification. Every individual generated 8 test cases once a time, but the test cases did not cover all paths necessarily. After selection and crossover operation, individuals will cover more and more paths; therefore the population will be better and better. This process will be terminated until the fitness = 1 has been found. For example, there is an array to record paths which has been covered. If one of the test cases covered path N0-N1-N3-N7, we add this path to the array, as shown in Fig. 8. The GA will not be terminated until all paths have been recorded. In other words, the process will be stopped before the best individual has been found, which saves the time and makes the algorithm more effective.

4 Experiments

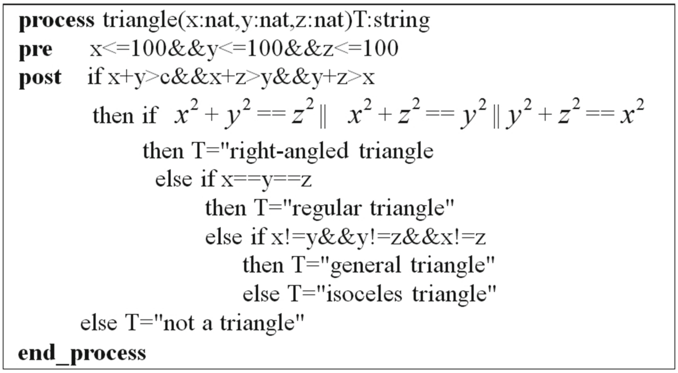
4.1 Experiment Method

Judging triangle's type is often introduced in the research about software testing as a typical example to demonstrate the idea of the testing. Firstly, we create the formal specification of judging triangle's type including precondition and post-condition, as shown in Fig. 9. There are three inputs and five paths in the program that implements the specification, as shown in Fig. 10. If the three inputs can not form a triangle, it is a test case that covers the path1. If the three inputs can form a triangle, they will cover one of the other four paths. All inputs are real number type. Due to the complexity and clarity of the logical in this program, even there are plenty of combinations of inputs, only a small amount of combinations can cover certain paths in the program. Therefore, blind search became costly.

The experiment environment parameters are shown in Table 1. Then we change the parameters in GA to determinate what the parameters set for. Figure 11 shows

Table 1. Experiment environment

OS	Windows 7
Processor	Intel CoreI i7 CPU 2.80 GHz
Memory(RAM)	8 GB
System type	64-bit operation system
Tool	Eclipse



```
process triangle(x:nat,y:nat,z:nat)T:string
pre     x<=100&&y<=100&&z<=100
post    if x+y>c&&x+z>y&&y+z>x
          then if  $x^2+y^2==z^2\|\ x^2+z^2==y^2\|y^2+z^2==x^2$
                then T="right-angled triangle
                  else if x==y==z
                        then T="regular triangle"
                        else if x!=y&&y!=z&&x!=z
                             then T="general triangle"
                             else T="isoceles triangle"
          else T="not a triangle"
end_process
```

Fig. 9. An example of a formal specification to judge triangle's type

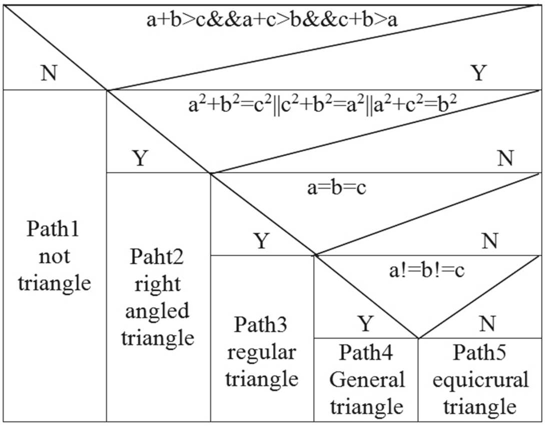


Fig. 10. An example of a paths map to judge triangle's type

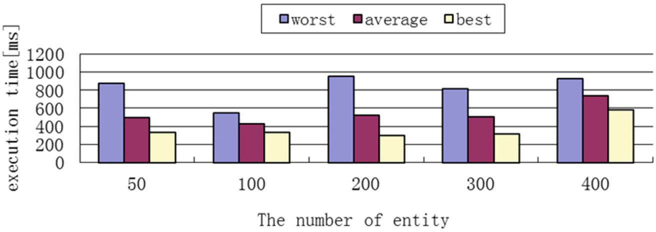


Fig. 11. Relationship between the execution time and the number of entity.

that if the number of the population is changed to 50, 100, 200 and 300, respectively, the execution time will be different. As the Fig. 11 shows that while the population size in 100 the experiment results are the stablest. From Fig. 12, we

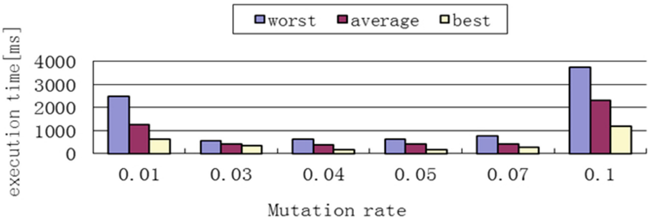


Fig. 12. Relationship between the execution time and a mutation rate.

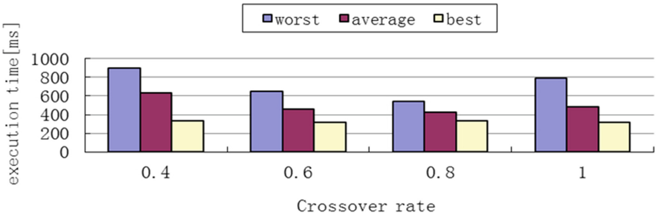


Fig. 13. Relationship between the execution time and a crossover rate.

can find that while the mutation rate is from 0.03 to 0.05 the cost of time is less. And the Fig. 13 shows cost of time is not very sensitive to crossover rate, in contrast, the mutation rate influenced the results more than other parameters.

Table 2 shows the execution parameters. The GA execution parameters are determined by preliminary tests and the choice of selection and crossover method. Based on the standard GA, we don not need to apply complicated selection and crossover method. In this phase, we prove that applying GA to automatically test cases generation is feasible and effective.

The design of the experiments is limited by the type of inputs. When the types of the input are naturel number, real number, or char, the design will be relatively easy. In this experiment, only inputs of naturel number type will be considered.

Table 2. GA execution parameters

No. of entities	100
Crossover rate	0.8
Mutation rate	0.03
Selection method	Tournament method
Crossover method	Two-point crossover
Mutation method	Simple mutation
Length of gene	35
Repeat time	50

Table 3. Execution time and generation

Execution Time			Generation			Path coverage rate
best	average	worst	best	average	worst	
121 ms	427 ms	940 ms	30	120	268	100 %

4.2 Experimental Results

Table 3 shows the results of the execution time and generations until a solution is reached by GA which repeats 50 times and all paths are covered. The results showed in Table 3 that the average generation is around 100, and the average time is less than 0.5 s.

4.3 Experiments About Comparison of GA and GA with Tabu List

The example containing 80 paths has been introduced here to show the difference between the standard GA and the GA with Tabu list. Figure 14 shows the formal specification and the paths we designed for the experiment.

Based on the formal specification, there are three inputs including variables x, y and z. However, it is not clear about how many paths in the relative program since the postcondition is true. We design a map of test paths based on the formal specification. In this map, there are 80 paths from start to end. As we marked the node in the map, every path has a sequence number when it passes through the nodes, which is showed at right Fig. 14. We generate test cases to cover the 80 paths in order to cover all paths in the relative program as far as possible. With the standard GA, we need to find the best individual who has covered all paths in the map; by using GA with Tabu list, the process will end when all paths have been traversed.

Table 4 shows the execution parameters in this experiment. The standard GA and GA with Tabu list have different parameters according to their different designs. As a result, when the number of entities is 750, we achieved a best result

Table 4. Exeperiment parameters

	GA	GA with Tabu list
2 No. of entities	750	50
Crossover rate	0.9	0.9
Mutation rate	0.03	0.03
Selection method	Tournament method	Tournament method
Crossover method	Two-point crossover	Two-point crossover
Mutation method	Simple mutation	Simple mutation
Length of gene	80	80
Repeat time	50	50

Precondition xy>z&&z!=0
postcondition true

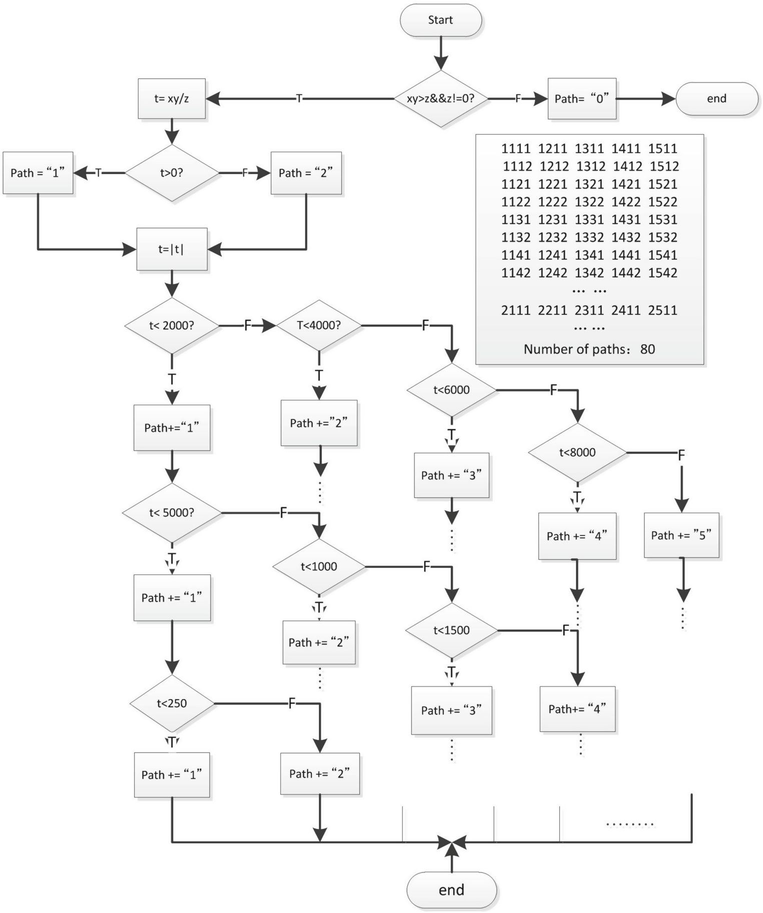


Fig. 14. An example of formal specification and a map of test paths

using standard GA; on the other hand, it costs too much time when the number of entities is large in the experiment using GA with Tabu list.

Table 5 shows the results of the two methods. When using standard GA, the execution will be terminated when the individual whose value of fitness is 1 has been found. It takes too much execution time to find the best individual who has covered all paths in the program. In practice, we merely need to cover all paths in the related program. The experiment of GA with Tabu list shows a much better result comparing with the standard GA. The GA with Tabu list has been terminated when all paths have been covered before finding the best one who has a best path coverage rate. From the Table 5, the GA with Tabu list has

Table 5. Comparison of GA and GA with Tabu list

	Average time	Average generation
GA	15033 ms	273.36
GA with Tabu list	11 ms	2.78

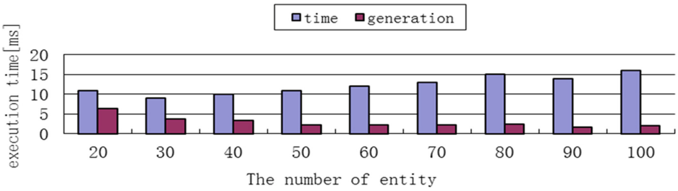


Fig. 15. Relationship between number of entities and execution time in the method using GA with Tabu list

saved much time and generation; that is because it did not cost too much time to find the best individual, and just several generations need to be produced to cover all paths in the related program.

Since the size of the population has influenced the results of the experiment using GA with Tabu list, we have changed the size of the population to do another experiment to see the relationship between the size of the population and the execution time of CPU. As is shown in Fig. 15, it got a best result around the population size of 30.

4.4 Discussion

Applying GA to automatic test cases generation can cover all paths in the test program effectively, as is shown in Table 6. The Vibration method which has been proposed covering all paths in the related program, but the results showed that the coverage rate is 92 % [12]. From Table 6, the V-method(Vibration method) has a much better result than Pairwise testing, however the coverage rate is still not reach the 100 %. The GA can cover all paths based on formal specification which is more effective than V-method and Pairwise testing method. On the other hand, this method need high cost for fitness evaluations of individuals because of the necessity for the running the program many times. Therefore, we need to speed up the proposed method using parallel processing on many-core architectures.

In order to improve the GA, we combined GA with Tabu search, and execution time has been reduced obviously, which is showed in Table 5. As shown in Table 5, the test case generation method using GA with Tabu list takes much less time than the standard GA. In standard GA, we attempt to find the best individual who covered all paths, on the contrary, the GA with Tabu list found all paths in the program before finding the best one. In the beginning, the two

Table 6. Comparison of path coverage rate

Testing methods	Number of test cases	Path coverage rate
Pairwise testing	58	53 %
V-Method	56	92 %
Standard GA	80	100 %
GA with Tabu list	80	100 %

methods are introduced to find the test cases which satisfy the test condition, afterwards, the standard GA focus on find the best one who covered all paths, while the GA with Tabu list does not care about the fitness function in the later time.

In this paper, we assume that the number of all paths in the related program is known. In the other hand, in practice, we do not know how many paths in the related program. Fortunately, there is an open source software [17] on the internet which can check out the number of paths in the program. It makes that generating the test cases to cover all paths in the test program with our method become practicable. We also need evaluations using a more complicated and practical program as our future work.

5 Conclusion

In this paper, we applied GA to generate test cases based on a formal specification. We demonstrated the procedure of the design about how to generate test cases using GA and our proposed method is more effective than conventional methods and can cover all paths. We also conducted two experiments to test our methods; one is to apply the standard GA to generate test cases of judging a triangle; the other containing 80 paths to compare the performance of GA and GA with Tabu list. The results show that the path coverage rate can reach 100 %, and the GA with Tabu list has a better performance than standard GA. How to locate the bugs in the program is still a problem unaddressed in this paper. Our future work will concentrate on detecting the bugs in the program using a more practical problem.

Acknowledgements. First and foremost, we would like to show my deepest gratitude to Prof. Liu, a respectable and responsible professor of the Department of Computer and Information Sciences in Hosei University, for their valuable comments and discussions on experimental results. This research is partly supported by the Hitachi Management Partner Corporation.

References

1. Horcher, H.-M.: Improving software tests using Z specifications. In: Proceeding 9th International Conference of Z Users, The Z Formal Specification Notation (1995)

2. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall, Englewood Cliffs (1992)
3. Offutt, J., Abdurazik, A.: Generating tests from uml specifications. In: France, R.B. (ed.) UML 1999. LNCS, vol. 1723, pp. 416–429. Springer, Heidelberg (1999)
4. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley Object Technology Series, Boston (1998)
5. Chang, J. Richardson, D.J.: Structural specification-based testing: automated support and experimental evaluation. In: Proceeding 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 285–302, Sept 1999
6. Sanker, S. Hayes, R.: Specifying and testing software components using ADL. Technical report SMLI TR-94-23, Cun Microsystems Laboratories, Inc., Mountain View, CA, April 1994
7. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automatically testing based on java predicates. In: ISSTA 2002 Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 123–133 (2002)
8. Marisa A. S'anchez: Specification-based Testing. pp. 12–47 1997
9. Marinov, D., Khurshid, S.: TestEra: a novel framework for automated testing of Java programs. In: Proceeding 16th IEEE International Conference on Automated Software Engineering, San Diego, CA, Nov. 2001
10. Jackson, D., Schechter, I., Shlyakhter, I.: ALCOA: The alloy constraint analyzer. In: Proceeding 22nd International Conference on Software Engineering(ICSE), Limerick, Ireland, June 2000
11. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. Technical report 01–12, Department of Computer Science, Iowa State University, Nov 2001
12. Liu, S., Nakajima, S.: A vibration method for automatically generating test cases based on formal specifications. In: Software Engineering Conference, 2011 18th Asia Pacific, pp. 73–80, 5–8 Dec. 2011
13. Liu, S., Nakajima, S.: A decompositional approach to automatic test cases generation based on formal specification. In: 4th IEEE International Conference on Secure Software Integration and Reliability Improvement, Singapore, pp. 147–155, IEEE CS Press, 9–11 June 2010
14. Reeves, C.R.: Modern Heuristic Techniques for Combinatorial Problems. Blackwell Scientific, Oxford (1993)
15. Beasley, J.E.: A genetic algorithm for the set covering problem. Eur. J. Oper. Res. **94**, 392–404 (1996)
16. Beasley, D., Bull, D.R., Martin, R.R.: An overview of genetic algorithms: Part I, fundamentals. Univ. Comput. **15**, 58–59 (1993)
17. http://www.kailesoft.cn (cited 7.7.2014)
18. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-Data generation using genetic algorithms. J. Softw.Test. Verif. Reliab. **9**, 263–282 (1999)

Development of a Software Tool to Support Traceability-Based Inspection of SOFL Specifications

Jinghua Zhang[1] and Shaoying Liu[2](✉)

[1] Graduate School of Computer and Information Sciences,
Hosei University, Tokyo, Japan
jinghua.zhang.6w@stu.hosei.ac.jp
[2] Department of Computer and Information Sciences,
Hosei University, Tokyo, Japan
sliu@hosei.ac.jp

Abstract. When developing a formal specification for a software project using the SOFL three-step modeling approach, it is essential to ensure the conformance relation between every two level specifications. Inspection is an important technique to verify the specifications. In this paper, we describe an inspection method through building traceability for rigorously verifying the conformance relation. The method consists of two steps: (1) traceability establishment and (2) inspection of the target specifications through the built traceability. We also provide some inspection strategies such as checklists based on SOFL features to help the inspector find errors and keep the consistency. Our tool provides a convenient interface to separate components in different specifications and save their relationships to keep the consistency. We describe the design and implementation of our supporting tool in this paper. A case study to inspect the specifications of a travel plan booking system is given to show how the proposed method can be applied in practice .

Keywords: SOFL · Specification · Traceability · Inspection · Conformance

1 Introduction

One of the primary problems in software projects is that the requirements documented in specifications may not be accurately and easily understood by the developers carrying out different tasks [1]. In general, requirements specifications need to be written by humans, and probably will be changed during the communication between customers and designers. Therefore, the target specifications stand a great chance to contain errors. Eliminating the errors in the early phase of a software project can produce a considerable positive effect on the overall cost of the project, and the reliability of the final software product [2]. Formal engineering methods have been recognized as an effective and efficient approach for developing large-scale software systems. One way to improve the quality of specifications and therefore the quality of the corresponding software program is to formalize specifications. We choose Structured Object-Oriented Formal Language (SOFL) for this purpose in this paper.

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 32–46, 2015.
DOI: 10.1007/978-3-319-17404-4_3

The SOFL method provides a three-step approach to developing formal specifications. Such a development is an evolutionary process, starting from an informal specification, to a semi-formal one, to finally a formal specification [1]. In the evolutionary process, the errors can be made because of inaccurate understanding of the requirements, incorrect uses of mathematical expressions, or wrong decisions in defining data or functions [3]. When changes take place on one level specification, it may require appropriate changes in the related specifications. However, how to keep the conformance between the specifications after the changes still remains an unaddressed problem. Our research mainly focuses on how to sustain the consistency between different level specifications and eliminate errors.

According to the IEEE standard [4], the purpose of an inspection is to detect and identify software product anomalies. An inspection is a systematic peer examination that verifies the software product conforms to applicable regulations, standards, guidelines, plans, specifications, and procedures. The inspector collects software engineering data like anomaly and effort data by using supporting documentations such as checklists to show what should be checked.

In this paper, we propose an inspection method through building traceability for rigorously verifying the conformance relation. This method mainly consists of two steps: (1) traceability establishment and (2) inspection of the target specifications through the built traceability. The first step is based on the structure and syntax of SOFL three-step specifications, corresponding items will be generated together in the evolutionary process. The checklists will be provided to help the inspector confirm whether to establish the traceability between two items in different specifications or not. The second step inspects the target specifications through the built traceability. Pair review is a useful way to check whether the traceability is correct or not by comparing the textual specifications and the Condition Data Flow Diagram (CDFD). Our supporting tool provides a convenient interface to separate components in different specifications and save their relationships to keep the consistency. Based on the correct syntax of components, our tool can get all items automatically to check whether the target specification fits user's requirements or not. We present a case study of the inspection method by describing how it is applied to inspect the specifications of a travel plan booking system to show the method's feasibility, and explore some potential challenges in using our supporting tool.

The rest of this paper is organized as follows. We introduce some basic concepts in Sect. 2. Section 3 mainly discusses the possible way to build the traceability and how to inspect the components through the traceability. We discuss the design and implementation of our supporting tool in Sect. 4. In Sect. 5, a case study is given to show how the proposed method can be applied in practice. Related work is introduced in Sect. 6. Finally, we give conclusions and point out future research directions in Sect. 7.

2 Basic Concepts

In this section, we first introduce SOFL and then some inspection strategies, such as checklists and pair reviews for inspecting SOFL specifications.

2.1 SOFL

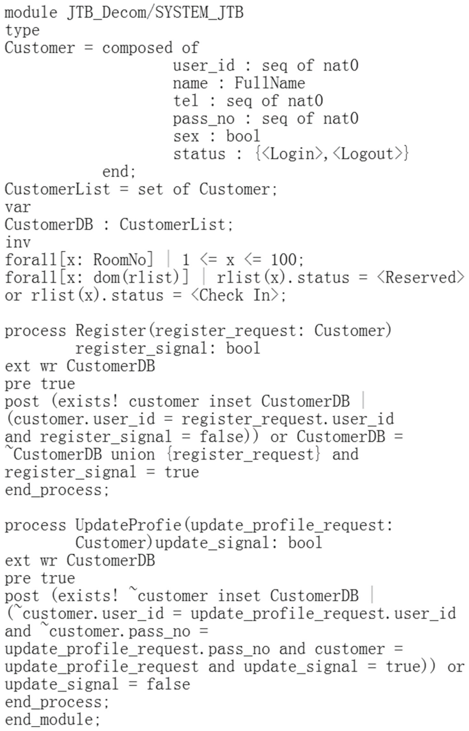
SOFL is a formal engineering method that provides a formal but comprehensible language for both requirements and design specifications. A SOFL specification mainly consists of two parts: the textual specification and the Condition Data Flow Diagram. The textual specification is a written documentation and mainly built by the component called “process”. A process models a transformation from input to output, which provides pre-condition and post-condition to describe the functionality and constraints of transferred data. Different processes contact with each other to handle data. A set of processes can be defined in a “module”, which can achieve some independent functions of the target system. At the same time, some processes can also be decomposed into a low level “module”, which can explain the complicated data manipulation more clearly. By combining the generation and decomposition of processes reasonably, we can easily achieve the system requirements in our SOFL specification. Figure 1 shows an example of SOFL textual specification.

The textual specification is produced based on a three-step approach to developing formal specifications. Such a development is an evolutionary process, starting from an informal specification, to a semi-formal specification, then to a formal specification. Informal specification is the first step in SOFL method to reach user’s requirements. Although it is difficult to define the concept of a well-organized specification, such a specification must clearly and concisely describe the following items:

(1) *the functions to be implemented in the software project;*
(2) *the resources to be used in implementing functions;*
(3) *the necessary constraints on both functions and resources.*

The semi-formal specification derives from the informal specification. Its goal is to clarify and define all the functions, resources, and constraints, and to determine the relationships among those three parts contained in the informal specification. The most distinct feature of a semi-formal specification is that the format of the specification obeys the syntax of the formal specification, but the pre- and post-conditions of all processes in modules are defined in a natural language in an organized manner. In the formal specification, by evolving all items from the semi-formal specification in logical expressions, some processes written by the natural language will be found too complicated to transform into logical expressions. Under this situation, we need decompose the process into some sub processes to keep them logical.

Another important part of SOFL is Condition Data Flow Diagram. Different from the textual specification, the CDFD is a directed graph that specifies how processes work together to provide functional behaviors. The process specification mainly focuses on the internal logical relationships and data constraints, while the CDFD mainly represents the relation between different processes by transferring different data. Figure 2 shows an example of CDFD describing a flight plan.



```
module JTB_Decom/SYSTEM_JTB
type
Customer = composed of
                   user_id : seq of nat0
                   name : FullName
                   tel : seq of nat0
                   pass_no : seq of nat0
                   sex : bool
                   status : {<Login>,<Logout>}
           end;
CustomerList = set of Customer;
var
CustomerDB : CustomerList;
inv
forall[x: RoomNo] | 1 <= x <= 100;
forall[x: dom(rlist)] | rlist(x).status = <Reserved>
or rlist(x).status = <Check In>;

process Register(register_request: Customer)
        register_signal: bool
ext wr CustomerDB
pre true
post (exists! customer inset CustomerDB |
(customer.user_id = register_request.user_id
and register_signal = false)) or CustomerDB =
~CustomerDB union {register_request} and
register_signal = true
end_process;

process UpdateProfie(update_profile_request:
        Customer)update_signal: bool
ext wr CustomerDB
pre true
post (exists! ~customer inset CustomerDB |
(~customer.user_id = update_profile_request.user_id
and ~customer.pass_no =
update_profile_request.pass_no and customer =
update_profile_request and update_signal = true)) or
update_signal = false
end_process;
end_module;
```

Fig. 1. SOFL textual specification

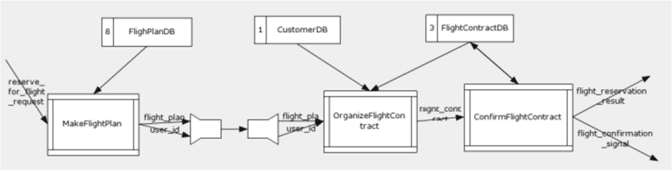


Fig. 2. CDFD describing a flight plan

From Fig. 2, we can see that each module generated by a set of processes has a corresponding CDFD. A process in the CDFD is presented as a rectangle pane and connects each other by arrows called data flows. A data flow represents input or output data in the textual specification. Also there is another kind of rectangles starting with a number called data stores. A data store is a variable holding data in rest. By using these rectangles, data can convert into the expected situation.

2.2 Checklists

When inspecting specifications, we need strategies to help inspectors check SOFL specifications easily. One strategy is Checklist. Checklists are a well-established reading support mechanism often used by individual inspectors for the purposes of preparation. Checklists are based upon a series of specific questions that are intended to focus the inspector's attention on common sources of defects.

The format of the checklist follows what is used by Laitenberger and DeBaud [5] and suggested by Chernak [6]. The schema consists of two components, "where to look" and "how to detect". The first component is a list of potential "problem spots" that may appear in the work product, and the second component is a list of hints on how to identify a defect in the case of each problem spot.

Finally, the questions are ordered to support the inspector in achieving a thorough understanding of the specifications. As the inspector moves through the different groups of questions (invariant, process, etc.), he successively proceeds from a higher-level and general perspective toward a more detailed and fine-grained one. Each group of questions requires more and more understanding of each item in three-step specifications, and the final question in the evolutionary specifications section, "does the target specification match the corresponding specification?" will be easier to answer once all the other questions have been applied.

2.3 Pair Review

Pair review is a group way of inspecting requirements specifications like a software development technique called pair programming. In pair programming, two programmers work together at a single keyboard, one is coding while the other observes and reviews. The roles are switched at regular intervals. Based on characters of SOFL language we mentioned above, pair review is very helpful when inspecting the textual specification and the corresponding CDFD. By reviewing the textual specification, we can see whether the input and output data in the CDFD are correct or not, and data should be stored in the right data stores. By reviewing the CDFD, we can check whether the set of processes in the corresponding textual specification are generated in the right order or not. At the same time, the types of data flow and the logical constraints about pre- and post-condition will be confirmed in the textual specifications.

3 Building Traceability and Inspection

Requirements traceability refers to the ability to describe and follow the life of a requirement, in both forwards and backwards direction. For example, from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases. Francisco et al. [7] think that requirements traceability refers to the ability to define, capture and follow the traces left by requirements on other elements of the software development environment and the trace left by those elements on requirements. Some traceability definitions emphasize the use of traceability to document the transformation of a requirement into successively concrete design and development artifacts. Elizabeth et al. [8] explain that in the requirements engineering field, traceability is about understanding how high-level requirements – objectives, goals, aims, aspirations, expectations, needs – are transformed into low-level requirements. It is therefore primarily concerned with the relationships between layers of information. From this definition, we can find that the SOFL method is from high-level requirements included in informal specification to low-level requirements included in formal specification. In this process, the requirements traceability is clear in corresponding items.

There are two steps in our inspection method through building traceability. First, we generate the traceability between three different specifications. The traceability means the congruent relationships of elements which represent the same users' requirements in different specifications. For example, a function in the informal specification may be correlated to a process in the corresponding semi-formal specification. Second, by comparing with the built traceability and CDFD, we inspect corresponding items in different specifications together.

Because there are three specifications, we separate the traceability into two parts to make it more clearly: (1) traceability between informal and semi-formal specifications, (2) traceability between semi-formal and formal specifications.

3.1 Building Traceability between Informal and Semi-formal Specifications

During the first part, user's requirements will be refined and described more precisely. To cover as many user's requirements as possible, the structures in the informal specifications are rough. They contain only three components: functions, data resources and constraints. Because of the partition in informal specification, the conversion to semi-formal specification is quite flexible and mainly depends on user's experience. However, we can still compare corresponding components based on structures in different specifications as shown in Fig. 3. By making a signal between the corresponding items – examples of components – in different specifications, every item will get the relationship with one or many items. We can make a checklist as shown in Table 1 to build the traceability about all items clearly between informal specification and semi-formal specification. If an item has no traceability with other items, the item should be removed or some items need to be added in the corresponding specification by comparing with the same kind of items.

3.2 Building Traceability between Semi-formal and Formal Specifications

For the second part, structures are almost the same between semi-formal and formal specifications. We should pay more attention about the invariants and processes. As shown in Table 2, corresponding invariant definitions will be compared with together to check whether their logical meanings are the same or not. Also we need to focus on the pre- and post-condition of processes to make the logical expression fit the requirements written in natural language.

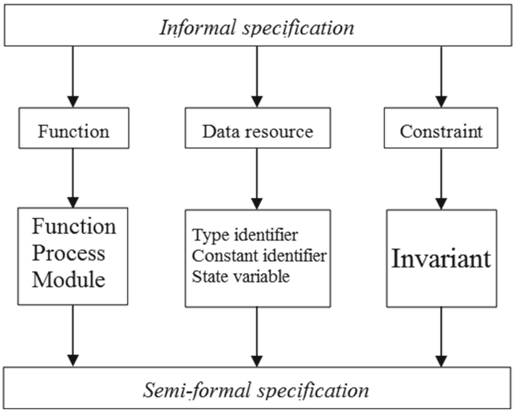


Fig. 3. Corresponding components between informal and semi-formal specifications

Table 1. Checklist about traceability between informal and semi-formal specifications

Informal specification(S1):		
	Component	*Question*
1	Function	Is the function too big to be separated into sub functions?
2		Does the function have the same name process in S2?
3	Data resource	Is the data resource used in the corresponding function?
4		Is the data resource evolved into the data type in S2?
5	Constraint	Is the constraint associated with a function or a data resource?
6		Does the constraint have the similar invariant in S2?
7		Is the constraint achieved in the pre- or post-condition of a process in S2?
Semi-formal specification(S2):		
8	Constant identifier	Is the constant identifier available and can be found in data resources in S1?
9	Type identifier	Is the type identifier needed from data resources in S1?
10	State variable	Is the state variable defined based on the type identifier?
11	Invariant	Does the invariant have the corresponding relation with the constraint in S1?
12	Process	Is the process named by the function in S1?
13		Is the process treated as a sub function in S1?

Table 2. Checklist about traceability between semi-formal and formal specifications

	Component	Question
Semi-formal specification(S2):		
1	Constant identifier	Is the constant identifier available and can be connected with similar constant in S3?
2	Type identifier	Is the type identifier defined based on the standard format?
3	State variable	Is the state variable defined based on the type identifier?
4	Invariant	Does the invariant match the meaning of the corresponding invariant in S3?
5	Process	Does the process have the same name, input data, output data in S3?
6		Does the pre- and post-condition of the process match the logical expression in S3?
7		Is the process divided into many sub processes in S3?
Formal specification(S3):		
8	Constant identifier	Is the constant identifier available and can be found in constant identifier declarations in S2?
9	Type identifier	Does the type identifier exist in S2?
10	State variable	Does the state variable exist in S2?
11	Invariant	Does the invariant have the same meaning of invariants in natural language in S2?
12	Process	Does the process have the same name, input data, output data in S2?
13		Does the pre- and post-condition of the process fit the natural expression in S2?
14		Is the process treated as a sub process and is a series of processes equal to the process in S2?

After generating two parts of traceability, we can inspect all items throughout the whole requirements specifications.

3.3 Inspection through Built Traceability

To inspect errors and defects, firstly we should provide the standard format of all data types. We can get them from the existing publication in [1], especially about the syntax of Set type, Sequence type, Composite type, Product type, Map type, Union type, and

Process type. The key words of these types will influence building traceability between different specifications.

For making pair reviews by two inspectors, the textual specification and corresponding CDFD should be inspected together. We can inspect the traceability based on building functional scenarios provided in [9]. Let P(Piv, Pov)[Ppre, Ppost] denote the formal specification of an operation P, where Piv and Pov are the sets of all input and output variables. Ppre and Ppost are the pre-condition and post-condition of operation P, respectively. Let $Ppost = C1 \wedge D1 \vee C2 \wedge D2 \vee \ldots \vee Cn \wedge Dn$, where Ci ($i \in \{1, 2, \ldots, n\}$) is a guard condition and Di is a defining condition. Then, a conjunction $\sim Ppre \wedge Ci \wedge Di$ is called a functional scenario.

To make inspectors to check corresponding items easily, we provide functional scenarios from CDFD as a standard to compare with both two specifications. In this situation, define the format (input_1, input_2,…, input_n){process_1, process_2,…, process_n}(output_1, output_2,…, output_n) as a functional scenario from CDFD. Fig. 4 shows a CDFD about making a hotel plan.

In this CDFD, we can get 3 functional scenarios:

(1) *(user_id, password, new_password){Login, ChangePassword} (password_success);*
(2) *(user_id, password){Login}(wrong_message);*
(3) *(user_id, password, hotel_request){Login, MakeHotelPlan}(hotel_plan).*

These three functional scenarios show three different conditions with submitting different data. When inspecting traceability between semi-formal and formal specification, a set of processes with same functional scenario (1), (2), (3) will be generated, and errors about wrong data flows should be removed.

4 Supporting Tool

We have built a supporting tool, called the *Traceability-based Specifications Inspection Supporting Tool* (TSIST), to support our inspection method through building traceability. The goal of building the tool is to help inspectors check specifications more precisely, and save the traceability information made by them for iterative inspections.

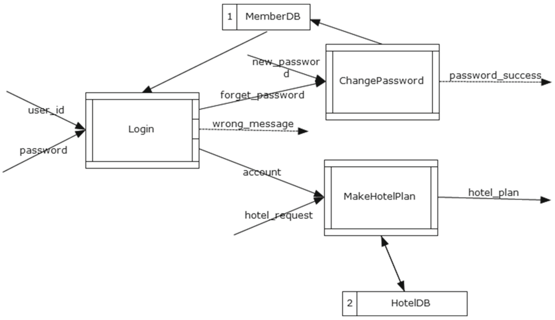


Fig. 4. CDFD of making a hotel plan

The supporting tool is implemented using Visual Studio 2012 with language C#. Figure 5 gives the CDFD of TSIST functions. As Fig. 5 shows, our tool can search key words from specifications, divide all items, then build traceability between different items in corresponding specifications, and finally inspect specifications by comparing traceability and CDFD. The whole process in using TSIST summarizes into three main functions below:

(1) Searching key words and inspecting syntax errors in three specifications;
(2) Selecting items from corresponding specifications manually or automatically;
(3) Building traceability between corresponding items in two specifications and saving traceability information for comparisons and iterative inspections.

4.1 Searching Key Words

TSIST provides the function to search key words in the specification documentation. When a key word is entered in the search column, our tool will match the key word in the target specification and find out all eligible elements. In this way, the user can check the syntax of target items quickly.

4.2 Selecting Items Automatically

To build traceability between two specifications, the inspector should get all items in textual specifications first. Obviously, we provide the manual way to add items directly. Based on the standard format of items, the inspector can also traverse the specification to get all items of the same component (such as Function, Data resource, Process).

To make the component "*process*" as an example. Setting the *targetComponent* = "*process*", *endFlag* = "*(*", *breakFlag* = ";", when reading the specifications from the beginning, the scanner gets the *targetComponent*, it will repeatedly read the next character until finding the key word *endFlag*. By using this way, the inspector can get names of all processes easily. From the structure and syntax of the component "process", we can find when we get the *breakFlag* before meeting the *endFlag*, it means the process ends with the syntax "*end_process;*" then the scanner will skip and try to find the next *targetComponent*. By changing *endFlag*, we can get input data, output data, pre- and post-condition of the target process respectively.

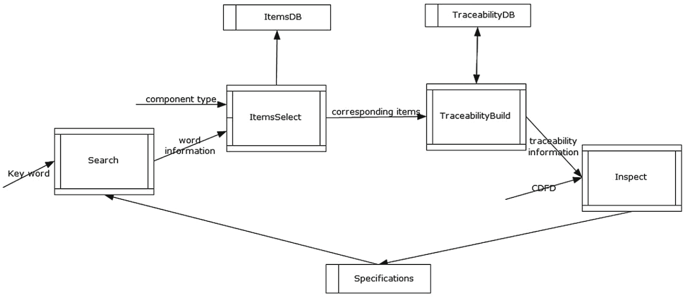


Fig. 5. CDFD of TSIST functions

4.3 Building Traceability

After generating all items in specifications, TSIST provide an interface to select corresponding items to build traceability between two specifications as shown in Fig. 6. At the same time, the traceability between informal and semi-formal specification and the traceability between semi-formal and formal specifications can be generated together to keep the consistency through the whole requirements specifications.

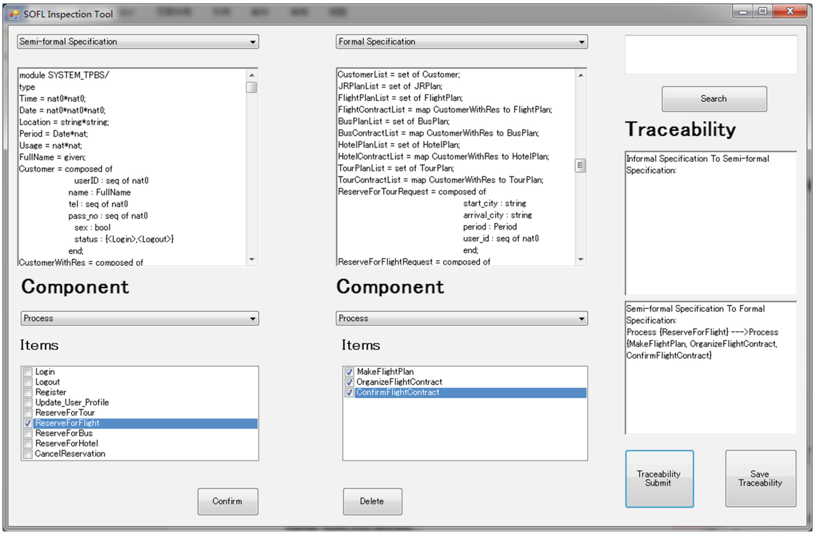
Compared with the checklists, the inspectors can build traceability by using our inspection supporting tool, the traceability between corresponding specifications can be saved and removed in iterative inspections. Through building traceability, the pair reviews based on the textual specification and CDFD will help inspectors check specifications and detect defects more precisely.

5 Case Study

We have conducted a case study applying our inspection method to the inspection of the specifications of a *travel plan booking system* (TPBS). The purpose of this case study is to show how our inspection method works through building traceability, to learn about its performance in terms of usability and capability of finding errors, and to investigate how the inspection method can be well supported by TSIST.

5.1 Background

TPBS specifications describe a travel plan booking system, which allows the customer to search travel information, design his personal travel plan, and book target services (flights, hotels, etc.). TPBS mainly includes four functions:



```
module SYSTEM_TPBS/
type
Time = nat0*nat0;
Date = nat0*nat0*nat0;
Location = string*string;
Period = Date*nat;
Usage = nat*nat;
FullName = given;
Customer = composed of
            userID : seq of nat0
          name : FullName
          tel : seq of nat0
          pass_no : seq of nat0
            sex : bool
            status : {<Login>,<Logout>}
          end;
CustomerWithRes = composed of
```

```
CustomerList = set of Customer;
JRPlanList = set of JRPlan;
FlightPlanList = set of FlightPlan;
FlightContractList = map CustomerWithRes to FlightPlan;
BusPlanList = set of BusPlan;
BusContractList = map CustomerWithRes to BusPlan;
HotelPlanList = set of HotelPlan;
HotelContractList = map CustomerWithRes to HotelPlan;
TourPlanList = set of TourPlan;
TourContractList = map CustomerWithRes to TourPlan;
ReserveForTourRequest = composed of
                                  start_city : string
                                  arrival_city : string
                                  period : Period
                                  user_id : seq of nat0
                                  end;
ReserveForFlightRequest = composed of
```

☐ Login
☐ Logout
☐ Register
☐ Update_User_Profile
☐ ReserveForTour
☑ ReserveForFlight
☐ ReserveForBus
☐ ReserveForHotel
☐ CancelReservation

☑ MakeFlightPlan
☑ OrganizeFlightContract
☑ ConfirmFlightContract

Informal Specification To Semi-formal Specification:

Semi-formal Specification To Formal Specification:
Process {ReserveForFlight} --->Process {MakeFlightPlan, OrganizeFlightContract, ConfirmFlightContract}

Fig. 6. Interface of TSIST

(1) *designing the tour plan;*
(2) *reserving flights;*
(3) *making bus arrangement;*
(4) *booking hotels.*

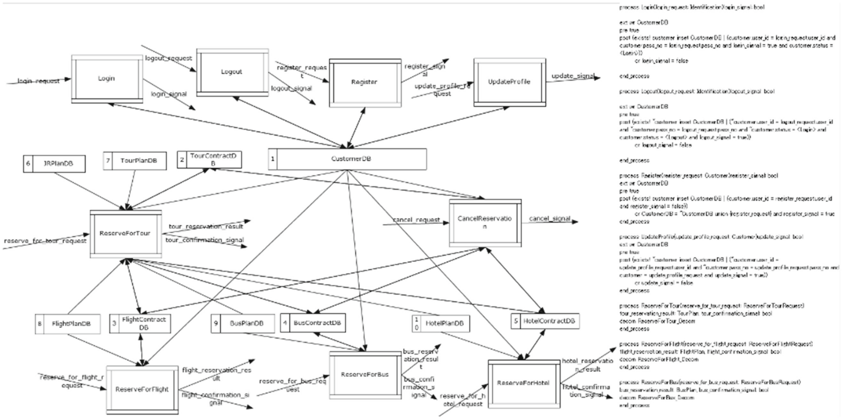
Figure 7 shows the textual specifications and corresponding CDFD of TPBS. From the informal specification to semi-formal specification, the functions of TPBS (such as Update_User_Profile, Reserve_ for_Hotel) are listed in details, showing how the input and out-put data flow between different processes. Data resources (such as Tour_Plan_Information, Bus_Plan_Information) will be checked to fit the types in the semi-formal specification. And constraints are used to show the range of data in the process. From the semi-formal specification to formal specification, the natural language used in pre- and post-condition evolves into logical expressions.

5.2 Building Traceability

As shown in Fig. 6, we can get all items such as "Process" in the target specification. Based on the checklist shown in Table 1, we decide where to build traceability between corresponding items. For example, in Fig. 6, we build the traceability between the process {ReserveForFlight} in the semi-formal specification and a set of the processes {MakeFlightPlan, OrganizeFlightContract, ConfirmFlightContract} in the formal specification because they realize the same functions. Table 3 gives the number of items and traceability in corresponding specifications. From this table, we can know every item has the traceability with others and the item in the high level may trace to a set of items in the more formal specification because they are more precise and smaller.

5.3 Inspection

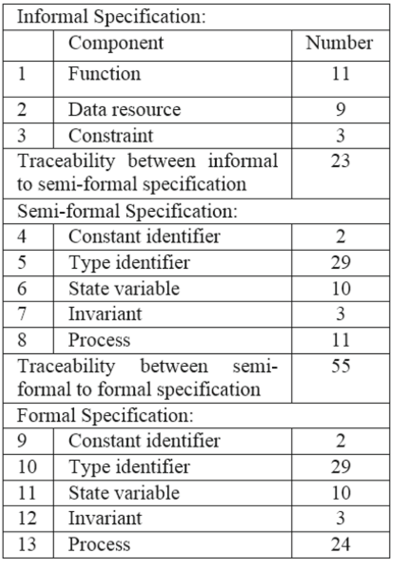
After building traceability, we can check corresponding items such as the process {ReserveForFlight} and the set of processes {MakeFlightPlan, OrganizeFlightContract,



```
process Login(login_request: Identification)login_signal: bool

ext wr CustomerDB
pre true
post (exists! customer inset CustomerDB | (customer.user_id = login_request.user_id and
customer.pass_no = login_request.pass_no and login_signal = true and customer.status =
<Login>))
        or login_signal = false

end_process

process Logout(logout_request: Identification)logout_signal: bool

ext wr CustomerDB
pre true
post (exists! ~customer inset CustomerDB | (~customer.user_id = logout_request.user_id
and ~customer.pass_no = logout_request.pass_no and ~customer.status = <Login> and
customer.status = <Logout> and logout_signal = true))
        or logout_signal = false

end_process

process Register(register_request: Customer)register_signal: bool
ext wr CustomerDB
pre true
post (exists! customer inset CustomerDB | (customer.user_id = register_request.user_id
and register_signal = false))
        or CustomerDB = ~CustomerDB union {register_request} and register_signal = true
end_process

process UpdateProfile(update_profile_request: Customer)update_signal: bool
ext wr CustomerDB
pre true
post (exists! ~customer inset CustomerDB | (~customer.user_id =
update_profile_request.user_id and ~customer.pass_no = update_profile_request.pass_no and
customer = update_profile_request and update_signal = true))
        or update_signal = false
end_process

process ReserveForTour(reserve_for_tour_request: ReserveForTourRequest)
tour_reservation_result: TourPlan, tour_confirmation_signal: bool
decom ReserveForTour_Decom
end_process

process ReserveForFlight(reserve_for_flight_request: ReserveForFlightRequest)
flight_reservation_result: FlightPlan, flight_confirmation_signal: bool
decom ReserveForFlight_Decom
end_process

process ReserveForBus(reserve_for_bus_request: ReserveForBusRequest)
bus_reservation_result: BusPlan, bus_confirmation_signal: bool
decom ReserveForBus_Decom
end_process
```

Fig. 7. Textual specifications and corresponding CDFD of TPBS

Table 3. Number of Items and Traceability



Informal Specification:		
	Component	Number
1	Function	11
2	Data resource	9
3	Constraint	3
Traceability between informal to semi-formal specification		23
Semi-formal Specification:		
4	Constant identifier	2
5	Type identifier	29
6	State variable	10
7	Invariant	3
8	Process	11
Traceability between semi-formal to formal specification		55
Formal Specification:		
9	Constant identifier	2
10	Type identifier	29
11	State variable	10
12	Invariant	3
13	Process	24

ConfirmFlightContract} together in the textual specifications. They will be checked whether they are equal not only in the syntax domain but also in the logical domain. Pair Review based on textual documentations and CDFD (also includes the decomposition of CDFD) helps inspectors understand requirements easily and find errors through data flows. For example, from the CDFD of the process "*ReserveForFlight*", we can know the process has the input "*reserve_for_flight_request*" and the output "*flight_reservation_result*", "*flight_confirmation_signal*". At the same time, this process has some operations with three data stores called "*FlightPlanDB*", "*FlightContractDB*", "*CustomerDB*", respectively. These features should be reflected in the corresponding textual specification. The incorrect data or missing processes in TPBS can be corrected in our supporting tool. The traceability through three specifications is saved for the iterative inspection.

6 Related Work

Many publications have affirmed that the requirements traceability plays an essential role in the software inspection. Although there still exists many challenges about how to create the traceability and how to inspect requirements based on the traceability.

Gotel and Finkelstein investigated and discussed the underlying nature of the requirements traceability problem in [10]. They introduced the distinction between pre-requirements specification traceability and post-requirements specification traceability, and pointed out that the major problems attributed to poor requirements traceability are due to inadequate pre-requirements specification traceability. They thought the solution is to re-focus research efforts on pre-requirements specification traceability from increasing awareness to access and presentation of information.

Francisco et al. [7] introduced a cited tool called TOOR to trace requirements considering both technical and social factors. TOOR can link requirements to design documents, specifications, code, and other artifacts in the life cycle through user-definable relations that are meaningful for the kind of connection being made by using both browsing and regular-expression search.

Letelier [11] presented a traceability metamodel integrating textual specifications with standard UML specifications, using the UML context itself. The metamodel offers a core framework for types of entities and types of traceability links that can be adapted to a particular UML project. Additionally, a configuration process for requirements traceability based on the corresponding UML profile was sketched.

Works made by Balasubramaniam [12] focused on the role of institutional context and the strategic conduct in explaining the differences in traceability practice across system development efforts. By contrasting two extremes of practice – low-end and high-end – they provided an understanding of these factors affecting traceability practice.

Corriveau [13] developed a process called Traceability for Object-Oriented Quality Engineering, or TOOQE, to minimize scaling problems caused by the quick increasing software development with the number of designers involved in a project. TOOQE emphasized traceability and the integration of development and testing to achieve quality and maintainability. TOOQE featured an iterative design process that let the developer correct mistakes and learn more about the problem they are trying to solve as they go along. Each iteration includes requirements capture, analysis, design, coding, and testing.

Based on the related work above, we know how to build the traceability faces many problems not only in the technical field but also in the social factors. We must provide enough information about requirements to support the traceability and save it in a proper way in the evolving process.

7 Conclusion and Future Work

In this paper, we propose an inspection method through building traceability for rigorously verifying the conformance relationship. This method mainly consists of two steps: (1) traceability establishment and (2) inspection of the target specifications through the built traceability. Inspection strategies such as Checklist and Pair review are used to help the inspectors build traceability and check textual specifications and CDFD together for finding errors.

Our supporting tool can divide all items based on the component types, build traceability and check specifications through traceability and CDFD. We present a case study applying our inspection method to inspect the specifications of a travel plan booking system, to show the method's feasibility and capability of finding errors, and to investigate how the inspection method can be well supported by our tool.

In the future, we will improve our supporting tool to make it more user-friendly and support more ways of building traceability and inspecting specifications.

References

1. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer, Heidelberg (2004)
2. Boehm, B.W.: Software Engineering Economics. Prentice Hall, Englewood Cliffs (1981)
3. Liu, S., McDermid, J., Chen, Y.: A rigorous method for inspection of model-based formal specifications. IEEE Trans. Reliab. **59**(4), 667–684 (2010)
4. IEEE: 1028-2008 IEEE Standard for Software Reviews and Audits. IEEE Computer Society (2008)
5. Laitenberger, O., DeBaud, J.-M.: An encompassing life cycle centric survey of software inspection. J. Syst. Soft. **50**(1), 5–31 (2000)
6. Chernak, Y.: A statistical approach to the inspection checklist formal synthesis and improvement. IEEE Trans. Softw. Eng. **22**(12), 866–874 (1996)
7. Francisco, P.A.C., Goguen, J.A.: An object-oriented tool for tracing requirements. IEEE Softw. **13**(2), 52–64 (1996)
8. Elizabeth, H., Jackson, K., Dick, J.: Requirements Engineering. Springer, New York (2005)
9. Li, M., Liu, S.: Automated functional scenario-based formal specification animation. In: 19th Asia-Pacific Software Engineering Conference, pp. 107–115. IEEE CS Press (2012)
10. Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: Proceedings of the IEEE First International Conference on Requirements Engineering, pp. 94–101 (1994)
11. Letelier, P.: A framework for requirements traceability in uml-based projects. In: 1st International Workshop on Traceability in Emerging Forms of Software Engineering (2002)
12. Balasubramaniam, R.: Factors influencing requirements traceability practice. Commun. ACM **41**(12), 37–44 (1998)
13. Corriveau, J.-P.: Traceability process for large OO projects. Computer **29** 63–68 (1996)

Model Checking and Animation

Unified Bounded Model Checking for MSVL

Bin Yu, Zhenhua Duan(✉), and Cong Tian

ICTT and ISN Lab, Xidian University, Xi'an 710071, P.R. China
yubin@stu.xidian.edu.cn, zhenhua_duan@126.com,
{zhhduan,ctian}@mail.xidian.edu.cn

Abstract. This paper presents Unified Bounded Model Checking (UBMC) for the verification of an infinite state system described with Modeling, Simulation and Verification Language (MSVL) which is an executable subset of Projection Temporal Logic (PTL). The desired property is specified by a Propositional PTL (PPTL) formula. We present the bounded semantics of PPTL and the approach to implementing UBMC. A Bounded Labeled Normal Form Graph (BLNFG) is constructed on the fly and a counterexample of minimal length is produced to ease the interpretation and understanding for debugging purposes. Finally, a resource allocation algorithm is presented as an example to illustrate how the proposed approach works.

Keywords: Bounded model checking · Unified model checking · Propositional Projection Temporal Logic · Modeling · Verification

1 Introduction

Techniques for automatic formal verification of finite state transition systems have been studied in recent years. Compared to other formal verification techniques (e.g. theorem proving), model checking [1,2] is an automatical approach. Model checking has been widely used in many fields such as verification of hardware, software and communication protocols. In model checking, the system to be verified is modeled as a finite state machine, and the specification is formalized by a temporal logic formula.

For a system in practice, the number of states in it can be very large and the explicit traversal of the state space becomes infeasible. To fight with this problem, several approaches, such as Symbolic Model Checking (SMC) [3], Abstract Model Checking (AMC) [4], and Compositional Model Checking [5], etc. have been proposed with success. The combination of symbolic model checking with BDDs [6,7] pushed the barrier to systems with 10^{20} states and more later [3]. But the bottleneck of SMC is the amount of memory that is required for storing and manipulating BDDs. The boolean functions required to represent the set of states can grow exponentially. Bounded model checking (BMC) is an important

This research is supported by the NSFC Grant No. 61133001, 61272117, 61322202, 61420106004, and 91418201.

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 49–61, 2015.
DOI: 10.1007/978-3-319-17404-4_4

progress in formal verification after SMC [8]. The basic idea in BMC is to search for a counterexample in executions whose length is bounded by some integer k. If the property is not satisfied, an error is found. Otherwise, we cannot tell whether the system satisfies the property or not. In this case, we can consider to increase k, and then perform the process of BMC again. BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods rather than BDDs. SAT procedures do not suffer from the state space explosion problem of BDD-based methods. Modern SAT solvers can handle propositional satisfiability problems with hundreds of thousands of variables or more. Tools supporting BMC are NuSMV2 [9], bounded model checker developed by CMU [10], Thunder of Intel [11], and so on.

With model checking and bounded model checking, the mostly used temporal logics are LTL [12], CTL [1], and their variations. However, expressiveness of both LTL and CTL is not powerful enough. There are at least two types of properties in practice which cannot (or with difficulty to) be specified by LTL and CTL: (1) time related properties such as "a property P holds after the 100^{th} time unit and before the 200^{th} time unit"; (2) periodically repeated of property P. The expressiveness of Propositional Projection Temporal Logic (PPTL) [13] is full regular [14] which allows us to verify full regular properties and time related properties of systems in a convenient way.

In recent years, the verification of infinite state systems has attained increasing interest. The main limitation of model checking is that it is restricted to (essentially) finite-state systems. In general, the model checking problem is undecidable for infinite state systems, and hence, it may happen that the verification process does not terminate. In the verification of an infinite state system, theorem proving [15] is a powerful technique. Predicate abstraction has been introduced as a technique for reduction of infinite state systems to finite one in the work of Graf and Saidi [16]. Verification by abstraction can be applied to infinite state systems as shown in [17–19]. Another way to deal with the difficulty of verification is the method of compositional verification that uses the combination of temporal case splitting and data type reductions to reduce types of infinite or unbounded range to small finite types, and arrays of infinite or unbounded size to small fixed-size arrays [20,21]. In bounded model checking of infinite state systems [22], three-valued logic is employed in order to explicitly forward uncertain information in the case a proof cannot be established due to insufficient bounds.

Modeling, Simulation and Verification Language (MSVL) is a subset of Projection Temporal Logic (PTL) [13,23] with framing techniques [24]. It can be used for the purpose of modeling, simulation and verification of software and hardware systems. For the verification of a finite system by MSVL, a method named Unified Model Checking has been presented in [25]. With this method, a system is first modeled as p in MSVL. Thus, p is a non-deterministic program of MSVL and also a formula of PTL. Second, the property we want to check is specified by a formula ϕ in PPTL. To check whether or not p satisfies ϕ amounts to proving $\models p \rightarrow \phi$. It turns out to prove $\not\models p \wedge \neg\phi$. Thus, for finite state programs in MSVL, we can translate the model checking problem into a satisfiability problem in PPTL since finite state programs in MSVL are equivalent to PPTL formulas. To check the satisfiability of $p \wedge \neg\phi$, Labeled Normal Form Graph (LNFG) of $p \wedge \neg\phi$ can be constructed. But for an infinite state transition system, the path in the LNFG may be a straight infinite line and we cannot get the result that whether the system satisfies the given property forever.

Given an infinite system p in MSVL, a property of the system in terms of a PPTL formula ϕ, and a user supplied upper bound k, we present an approach named Unified Bounded Model Checking (UBMC) which combines bounded model checking and unified model checking approaches. In order to do this, bounded semantics of PPTL is presented. Further, the procedure of UBMC can be described as a process to construct the Bounded Labeled Normal Form Graph (BLNFG) of $p \wedge \neg\phi$ on the fly. BLNFG is constructed progressively as the current bound increases. If a finite or an infinite counterexample is found at a given bound that is less than the upper bound, the construction of the BLNFG stops and the counterexample is output. When there is no new node to be dealt with and no counterexample is found, the construction of BLNFG terminates and the result is given that the property is valid. If the current bound is increasing until the upper bound with no counterexamples found, it cannot be determined whether the system satisfies the property or not. At this time, we can increase the upper bound and construct the BLNFG of $p \wedge \neg\phi$ again.

The main advantages of our technique are the follows. First, our method can partially verify an infinite system described by MSVL. We can give the result that whether the property is valid in bound k. Second, our method can find counterexamples relatively quicker. This is due to the depth first nature in the construction of our BLNFG. Finding counterexamples is arguably the most important feature of model checking. Third, it finds a counterexample of minimal length. This feature helps users to understand a counterexample more easily.

This paper is organized as follows. In the next section, as a property specification language, PPTL formulas are presented. Then the language MSVL used for the description of an infinite system is formalized. In Sect. 3, the bounded semantics of PPTL formulas is given. Next, the method for constructing a BLNFG is formalized in detail. A resource allocation algorithm is presented to illustrate how our approach works in Sect. 5. Finally, conclusion is drawn in Sect. 6.

2 Preliminaries

2.1 Propositional Projection Temporal Logic

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

$$P ::= p \mid \bigcirc P \mid \neg P \mid P_1 \vee P_2 \mid (P_1, \cdots P_m)\ prj\ P$$

where $p \in Prop$, $P_1, \cdots, P_m$ and P are all well-formed PPTL formulas. $\bigcirc$ (*next*) and prj (*projection*) are basic temporal operators.

A mapping from $Prop$ to $B = \{true, false\}$ is used to define a state s, $s : Prop \rightarrow B$. $s[p]$ denotes the valuation of p at the state s. An interval σ is a non-empty sequence of states. The length of σ, $|\sigma|$, is the number of states minus 1 if σ is finite, and ω otherwise. The set of non-negative integers N_0 with $\{\omega\}$, $N_\omega = N_0 \cup \{\omega\}$ is used for both finite and infinite intervals. The relational operators, $=, <, \leq$, is extended to N_ω by considering $\omega = \omega$, and for all $i \in N_0, i < \omega$. Moreover, the relation symbol $\preceq$ is defined as $\leq -(\omega, \omega)$.

In an interpretation $\mathcal{I} = (\sigma, i, j)$, σ is an interval, i an integer, and j an integer or ω such that $i \preceq j \leq |\sigma|$. If formula P is interpreted and satisfied over a subinterval $< s_i, \cdots, s_j >$ of σ with the current state being s_i, it is denoted by the notation $(\sigma, i, j) \models P$. The satisfaction relation ($\models$) is inductively defined as follows:

$I - prop$ $\mathcal{I} \models p$ iff $s_i[p] = true$, and $p \in Prop$ is an proposition
$I - not$ $\mathcal{I} \models \neg P$ iff $\mathcal{I} \not\models P$
$I - or$ $\mathcal{I} \models P \vee Q$ iff $\mathcal{I} \models P$ or $\mathcal{I} \models Q$
$I - next$ $\mathcal{I} \models \bigcirc P$ iff $i < j$ and $(\sigma, i+1, j) \models P$
$I - prj$ $\mathcal{I} \models (P_1, \cdots, P_m)\ prj\ P$, if there exist integers $r_0 \leq r_1 \leq \cdots \leq r_m \leq j$
such that $(\sigma, r_0, r_1) \models P_1$, $(\sigma, r_{l-1}, r_l) \models P_l, 1 < l \leq m$, and $(\sigma', 0, |\sigma'|) \models Q$
for one of the following σ':
(a) $r_m < j and \sigma' = \sigma \downarrow (r_0, \cdots, r_m) \cdot \sigma_{(r_m+1, \cdots, j)}$, or
(b) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \cdots, r_h)$ for some $0 \leq h \leq m$.

where P, $P_1, \cdots, P_m$ and Q are PPTL formulas.

A formula P is satisfied by an interval σ, denoted by $\sigma \models P$, if $(\sigma, 0, |\sigma|) \models P$. A formula P is called satisfiable if $\sigma \models P$ for some σ. A formula P is valid, denoted by $\models P$, if $\sigma \models P$ for all σ.

For any PPTL formula Q, it can be rewritten into its normal form [26]:

$$NF(Q) \equiv \bigvee_{j=0}^{n_0} (Q_{ej} \wedge empty) \vee \bigvee_{i=0}^{n} (Q_{ci} \wedge \bigcirc Q_{fi})$$

where Q_{ej} and Q_{ci} are conjunctions of atomic propositions (or their negations) in Q_p which is the set of atomic propositions appearing in the PPTL formula Q, and Q_{fi} is an arbitrary PPTL formula.

For a PPTL formula Q, its corresponding LNFG can be constructed, which explicitly illustrates models of the formula. Here, an example is given to show the LNFG of a PPTL formula intuitively and the formal definition can be found in [27].

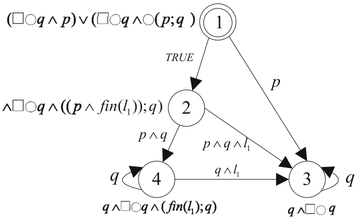


Fig. 1. LNFG of formula $\neg(true; \neg \bigcirc q) \wedge (p \vee \bigcirc(p; q))$

Example 1. LNFG of formula $\neg(true; \neg\bigcirc q) \wedge (p \vee \bigcirc(p; q))$ is shown in Fig. 1.

In the LNFG of a formula as shown in Fig.1, each node is specified by a PPTL formula, while each edge is a directed arc labeled with a state formula. The extra propositions l_k are employed to mark the infinite paths in the LNFG which are not the models of the PPTL formula.

2.2 Modeling, Simulation and Verification Language

MSVL is a subset of Projection Temporal Logic [13,23] with framing technique [24]. Based on the language, we have developed a model checking tool named MSV which works in three modes: modeling, simulation and verification.

The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

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

where n is an integer and x a variable. The elementary statements in MSVL are defined as follows. The meanings of all statements in MSVL are given in [13].

Termination: $empty$ *State Assignment*: $x <== e$ *Assignment*: $x := e$
State Frame: $lbf(x)$ *Interval Frame*: $frame(x)$ *Conjunction*: $p \wedge q$
Selection: $p \vee q$ *Next*: $\bigcirc p$ *Always*: $\Box p$ *Sequence*: $p; q$
Conditional: $if\ b\ then\ p\ else\ q \overset{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
While: $while\ b\ do\ p \overset{\text{def}}{=} (p \wedge b)^* \wedge \Box(empty \rightarrow \neg b)$

where x denotes a variable, e stands for an arbitrary arithmetic expression, b denotes a boolean expression, and p, q stand for programs of MSVL.

Any MSVL program p can be rewritten into its normal form [13,24]. According to normal form, we can construct an LNFG $G = (CL(p), EL(p), v_0, \mathbb{L} = \{\mathbb{L}_1, \ldots, \mathbb{L}_m\})$ to model an infinite state MSVL program p. Each node is specified by a program in MSVL, while each edge is a directed arc labeled with a state formula p_e from node q to node r and identified by a triple, (q, p_e, r).

Note that the number of nodes is finite only when the range of values of the variables in the program is limited to a finite set. When the range of variables is infinite, we cannot construct a finite LNFG of the program. For example, in the program $frame(i)\ and\ (int\ i <== 0\ and\ skip;\ while(true)\{(i := i + 1)\})$, the value of i is increasing and a finite LNFG of the program cannot be constructed always. In the LNFG of an infinite state MSVL program, there exist three kinds of paths: finite paths, loop paths and infinite paths with infinite states.

3 Bounded Semantics for PPTL

The basic idea of bounded model checking, as explained before, is to consider only a finite prefix of a path that may be a witness to the desired property. We

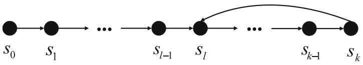


Fig. 2. σ is a (k, l)-loop

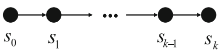


Fig. 3. σ is a finite interval

restrict the length of the prefix by some bound k. In practice, we progressively increase the bound, looking for witnesses in longer and longer traces.

A crucial observation is that, though the prefix of a path is finite, it still might represent an infinite path if there is a back loop from the last state of the prefix to any of the previous states as in Fig. 2. If there is no such loop, as in Fig. 3, the prefix does not say anything about the behavior of the path beyond state s_k.

Definition 1 ***((k,l)-loop).*** *For $l, k \in N_0$ and $l \leq k$, if there is a transition from s_k to s_l in σ i.e. $\sigma = (s_0, \cdots, s_{l-1}) \cdot (s_l, \cdots, s_k)^\omega$, we call interval σ a (k, l)-loop, k-loop for short.*

Obviously, if σ is an infinite interval with a loop, it must be a k-loop for some $k \in N_0$. We will use the notion of k-loop in order to define the bounded semantics of PPTL. The bounded semantics is an approximation to the unbounded semantics, which will allow us to define the bounded model checking problem. Since each PPTL formula can be transformed into an equivalent formula in NF, we do not need to deal with all types of PPTL formulas in the bounded semantics.

In the bounded semantics, we only consider a finite prefix of a path. In particular, we only use the first $k+1$ states $(s_0, \ldots, s_k)$ of a path to determine the validity of a formula along the path. If a path is a k-loop then we simply maintain the original semantics of atomic propositions, $\neg$, $\vee$, and $\bigcirc$ operators, because all the information about this infinite path is contained in the prefix of length k. Since *empty* $\equiv \neg \bigcirc$ *true* and *more* $\equiv \bigcirc$*true*, the bounded semantics of *empty* and *more* can be deduced by the bounded semantics of $\neg$ and $\bigcirc$. In fact, the formula *empty* cannot be satisfied over an infinite interval, while *more* is satisfied all the time in an infinite interval.

Definition 2 ***(Bounded Semantics for a Loop).*** *Let $k \in N_0$ and σ be a k-loop interval, a PPTL formula f is valid along σ with bound k (denoted by $\sigma \models_k f$) iff $\sigma \models f$.*

We now consider the case where σ is not a k-loop. We use the notation $(\sigma, i) \models_k f$ $(0 \leq i \leq k \leq |\sigma|)$ to represent that formula f is interpreted and satisfied over the subinterval $< s_i, \cdots, s_k >$ of σ with the current state being s_i. $(\sigma, 0) \models_k f$ is denoted by $\sigma \models_k f$.

In the bounded semantics without a loop, we only consider formulas constructed from atomic propositions and negations of atomic propositions with $\vee$, $\wedge$, and $\bigcirc$ operators as well as *empty* and *more*.

Definition 3 ***(Bounded Semantics without a Loop).*** *Let $k \in N_0$ and σ be an interval that is not a k-loop. The bounded satisfaction relation $\models_k$ is defined as follows:*

$$\begin{array}{lll}
(\sigma,i)\models_k p & \textit{iff} & s_i[p]=\textit{true if } p\in \textit{Prop is an atomic proposition}\\
(\sigma,i)\models_k \neg p & \textit{iff} & s_i[p]=\textit{false if } p\in \textit{Prop is an atomic proposition}\\
(\sigma,i)\models_k P_1\vee P_2 & \textit{iff} & (\sigma,i)\models_k P_1 \textit{ or } (\sigma,i)\models_k P_2\\
(\sigma,i)\models_k P_1\wedge P_2 & \textit{iff} & (\sigma,i)\models_k P_1 \textit{ and } (\sigma,i)\models_k P_2\\
(\sigma,i)\models_k \bigcirc P & \textit{iff} & i+1\le k \textit{ and } (\sigma,i+1)\models_k P\\
(\sigma,i)\models_k \textit{empty} & \textit{iff} & i=|\sigma|\\
(\sigma,i)\models_k \textit{more} & \textit{iff} & i<|\sigma|
\end{array}$$

Lemma 1. *Let $k\in N_0$, f be a PPTL formula, and σ a finite interval. We have $\sigma\models_k f\Rightarrow \sigma\models f$.*

Proof: To prove *Lemma* 1, we first prove a stronger conclusion given below:

$$(\sigma,i)\models_k f\Rightarrow(\sigma,i,|\sigma|)\models f\quad(0\le i\le k)$$

Lemma 1 can be concluded by setting $i=0$. We prove the above conclusion by induction on the structure of formula f:

Base case:

$f\equiv p\in \textit{Prop}:(\sigma,i)\models_k p\Rightarrow s_i[p]=\textit{true}\Rightarrow(\sigma,i,|\sigma|)\models p$

$f\equiv\neg p\in \textit{Prop}:(\sigma,i)\models_k\neg p\Rightarrow s_i[p]=\textit{false}\Rightarrow(\sigma,i,|\sigma|)\not\models p\Rightarrow(\sigma,i,|\sigma|)\models\neg p$

$f\equiv \textit{empty}:(\sigma,i)\models_k \textit{empty}\Rightarrow i=|\sigma|\Rightarrow(\sigma,i,|\sigma|)\models \textit{empty}$

Inductive cases: Suppose for any PPTL formula f, $(\sigma,i)\models_k f\Rightarrow(\sigma,i,|\sigma|)\models f$.

1. By hypothesis, when $i<k\le|\sigma|$, we have $(\sigma,i+1)\models_k f\Rightarrow(\sigma,i+1,|\sigma|)\models f$. By the definitions of semantics, $(\sigma,i+1)\models_k f$ iff $(\sigma,i)\models_k\bigcirc f$ and $(\sigma,i+1,|\sigma|)\models f$ iff $(\sigma,i,|\sigma|)\models\bigcirc f$, so we can get $(\sigma,i)\models_k\bigcirc f\Rightarrow(\sigma,i,|\sigma|)\models\bigcirc f$. When $i=k$, $(\sigma,i+1)\models_k f$ is *false*. Because $\textit{false}\Rightarrow(\sigma,i,|\sigma|)\models\bigcirc f$, we can get $(\sigma,i)\models_k\bigcirc f\Rightarrow(\sigma,i,|\sigma|)\models\bigcirc f$.
2. By hypothesis, we have $(\sigma,i)\models_k P_1\Rightarrow(\sigma,i,|\sigma|)\models P_1$ and $(\sigma,i)\models_k P_2\Rightarrow(\sigma,i,|\sigma|)\models P_2$. By the definitions of bounded semantics, we can easily get $(\sigma,i)\models_k P_1\vee P_2\Rightarrow(\sigma,i,|\sigma|)\models P_1\vee P_2$. Similarly, $(\sigma,i)\models_k P_1\wedge P_2\Rightarrow(\sigma,i,|\sigma|)\models P_1\wedge P_2$.

Note that we do not need to deal with *more* in the above inductive cases since $\textit{more}\equiv\bigcirc\textit{true}$.

Lemma 2. *Let f be a PPTL formula and σ a finite interval. Then $\sigma\models f\Rightarrow\exists k,k\in N_0,\sigma\models_k f$.*

Proof: Since σ is a finite interval, so $|\sigma|\in N_0$.

$$\begin{aligned}\sigma\models f&\Rightarrow(\sigma,0,|\sigma|)\models f\\&\Rightarrow(\sigma,0,k)\models f\wedge k=|\sigma|\\&\Rightarrow\exists k,k\ge0,(\sigma,0)\models_k f\\&\Rightarrow\exists k,k\ge0,\sigma\models_k f\end{aligned}$$

4 Unified Bounded Model Checking of MSVL

In the Unified Model Checking implemented in [25], LNFG of $p \wedge \neg\phi$ is constructed to check whether a finite state MSVL program p satisfies a PPTL formula ϕ. According to [27], finite and infinite models of $p \wedge \neg\phi$ are precisely characterized by the paths which satisfy certain conditions in the LNFG. For a PPTL formula Q, an interval σ_π can be defined for a given path π in the LNFG of formula Q and given a model σ of formula Q , $\sigma \models Q$, a path π_σ can be constructed according to the transition rules [27].

In the previous section, we defined the bounded semantics of PPTL. According to it, a BLNFG can be constructed to describe the model of $p \wedge \neg\phi$ in bound k.

Definition 4. ***(Bounded Labeled Normal Form Graph, BLNFG).*** *For a MSVL program p, a PPTL formula ϕ, and $k \in N_0$, its BLNFG is a tuple $G = (CL(p \wedge \neg\phi), EL(p \wedge \neg\phi), v_0, \mathbb{L} = \{\mathbb{L}_1, \ldots, \mathbb{L}_m\}, \mathbb{C} = \{\mathbb{C}_1, \ldots, \mathbb{C}_k\})$, where $CL(p \wedge \neg\phi), EL(p \wedge \neg\phi), V_0$ and $\mathbb{L}$ are identical to the ones defined in* [27] and each $\mathbb{C}_i \subseteq CL(p \wedge \neg\phi), 0 \leq i \leq k$, is the set of nodes with $c_q = i$, where c_q represents the depth of a node q.

Since the set $CL(p\wedge\neg\phi)$ of nodes and the set $EL(p\wedge\neg\phi)$ of edges are inductively produced by repeatedly rewriting the new created nodes into their normal forms, the BLNFG can be constructed progressively with the current bound increasing until a user supplied upper bound k. When constructing BLNFGs by normal form reductions, for any chop formula $P;Q$, we equivalently rewrite it by $P \wedge fin(l_k);Q$ as implemented in [27]. For convenience, we use $\mathsf{inf}(\pi)$ to denote the set of nodes which infinitely often occur in the infinite path π.

By conclusions in [27], we can get the Corollaries 1 and 2 respectively as follows.

Corollary 1. *If π is a finite or an infinite path with $\mathsf{inf}(\pi) \nsubseteq \mathbb{L}_i$ for all $1 \leq i \leq m$ and $c_q \leq k$ $(k \in N_0)$ for all nodes q on π in the BLNFG of formula $p \wedge \neg\phi$, then the interval obtained from the path $\sigma_\pi \models_k p \wedge \neg\phi$.*

Corollary 2. *For a finite or an infinite interval σ, if $\sigma \models_k p \wedge \neg\phi$, then π_σ translated from σ with $\mathsf{inf}(\pi_\sigma) \nsubseteq \mathbb{L}_i$ for all $1 \leq i \leq m$ and $c_q \leq k$ for all nodes q on π_σ can be found in the BLNFG of formula $p \wedge \neg\phi$.*

Corollaries 1 and 2 tell us that a finite or an infinite interval $\sigma \models_k p \wedge \neg\phi$ iff there exists a corresponding finite or infinite path satisfying certain conditions in the BLNFG of formula $p \wedge \neg\phi$ in bound k.

Because a MSVL program p does not satisfy a PPTL formula ϕ iff $p \wedge \neg\phi$ is satisfiable, then we can get the following corollary which is important in our unified bounded model checking.

Corollary 3. *In the LNFG of an infinite system p in MSVL, finite paths precisely characterize finite models of p; loop paths with $\mathsf{inf}(\pi) \nsubseteq \mathbb{L}_i$ for all $1 \leq i \leq m$, precisely characterize loop models of p.*

According to Corollary 3, our bounded model checking approach can be deduced to the construction of the BLNFG of $p \wedge \neg\phi$. In the on-the-fly construction of $p \wedge \neg\phi$, i is the current bound which stops increasing once a counterexample is found or reaches the upper bound. Initially, we create the root node $p \wedge \neg\phi$ and set $c_{p\wedge\neg\phi}$ to 0.

For a given bound i, nodes whose depth equals i are dealt with. In order to retain consistency, we use $p \wedge \neg\phi$ to represent the node that will be dealt with. For a node $p \wedge \neg\phi$, p and $\neg\phi$ are rewritten into their normal forms respectively. The function $NF()$ defined in [26] is called to produce normal form of a PPTL formula or MSVL program. Then we can get the conjunction of these two normal forms. The first part of the conjunction is a disjunction. A finite path ended by the empty node is found once one disjunct can be satisfied. At this time, the construction of the BLNFG terminates and a finite counterexample is output. Because the depth of nodes that are dealt with is increasing progressively, our process produces counterexamples of minimal length, which eases the understanding for debugging purposes. If no counterexample is found by checking the

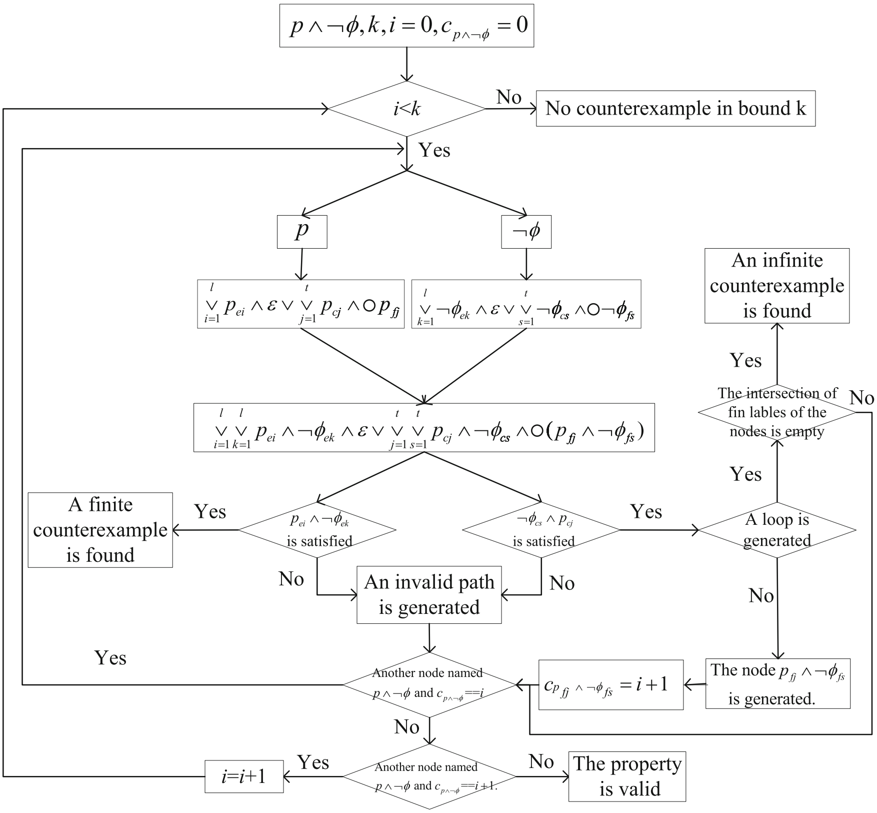

$p \wedge \neg\phi, k, i=0, c_{p\wedge\neg\phi}=0$
$i<k$
No
No counterexample in bound k
Yes
p
$\neg\phi$
$\bigvee_{i=1}^{l} p_{ei} \wedge \varepsilon \vee \bigvee_{j=1}^{t} p_{cj} \wedge \bigcirc p_{fj}$
$\bigvee_{k=1}^{l} \neg\phi_{ek} \wedge \varepsilon \vee \bigvee_{s=1}^{t} \neg\phi_{cs} \wedge \bigcirc \neg\phi_{fs}$
An infinite counterexample is found
Yes
The intersection of fin lables of the nodes is empty
No
$\bigvee_{i=1}^{l}\bigvee_{k=1}^{l} p_{ei} \wedge \neg\phi_{ek} \wedge \varepsilon \vee \bigvee_{j=1}^{t}\bigvee_{s=1}^{t} p_{cj} \wedge \neg\phi_{cs} \wedge \bigcirc(p_{fj} \wedge \neg\phi_{fs})$
Yes
A finite counterexample is found
Yes
$p_{ei} \wedge \neg\phi_{ek}$ is satisfied
$\neg\phi_{cs} \wedge p_{cj}$ is satisfied
Yes
A loop is generated
No
An invalid path is generated
No
No
Yes
Another node named $p \wedge \neg\phi$ and $c_{p\wedge\neg\phi}==i$
$c_{p_{fj} \wedge \neg\phi_{fs}} = i+1$
The node $p_{fj} \wedge \neg\phi_{fs}$ is generated.
No
i=i+1
Yes
Another node named $p \wedge \neg\phi$ and $c_{p\wedge\neg\phi}==i+1$.
No
The property is valid

Fig. 4. The construction of the BLNFG for $p \wedge \neg\phi$

first part, the second part of the conjunction will be dealt with. For every disjunct, a new node will be generated if the present state can be satisfied. If the node $p_{fj} \wedge \neg\phi_{fs}$ does not exist, a new node $p_{fj} \wedge \neg\phi_{fs}$ whose depth is $i+1$ is generated. Otherwise, a loop will be generated. If $\mathsf{inf}(\pi) \nsubseteq \mathbb{L}_i$ for all $1 \leq i \leq m$, the infinite path is output as a counterexample and the construction of BLNFG terminates. If not, this infinite interval is not a model of $p \wedge \neg\phi$ and is not taken into account. At this time, we will check whether there exist other nodes whose depth is equal to i. If these nodes exist, they will be dealt with by the same process above. Those new generated nodes whose depth is equal to $i+1$ will not be dealt with immediately. When the current bound increases to $i+1$, $p_{fj} \wedge \neg\phi_{fs}$ will be considered.

In case that all nodes whose depth is i have been dealt with and no counterexample has been found, the current bound i will increase. If there is no node to be dealt with at this time, it means that the whole LNFG of $p \wedge \neg\phi$ has been constructed. Because no valid path exists in the LNFG, we can get the result that the property ϕ is valid. If the value of i is still less than the upper bound k, the nodes whose depth equals i will be dealt with by the same process above. If the value of i is larger than the upper bound k, the process has to stop and it cannot be determined whether the system satisfies the property or not. The construction of the BLNFG for $p \wedge \neg\phi$ is depicted in Fig. 4.

Based on our UBMC algorithm, the model checker acting as a module in the MSV toolkit [25] has been developed. Under the bounded verification mode, a finite or an infinite system model is described by a MSVL program and the property is specified by a PPTL formula. A upper bound can be set by the user, otherwise it will be the default value.

5 A Case Study: Verification of Resource Allocation Algorithm

Banker's algorithm [28] is a resource allocation and deadlock avoidance algorithm developed by Dijkstra. It tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources. The algorithm tests for possible deadlock conditions for all other pending activities before deciding whether allocation is permitted.

The algorithm is originally developed in the design process of operating systems. When a new process enters a system, it must declare the maximum number of each resource type that it may ever claim. When a process gets all its requested resources, it must return them in a finite time interval. Many variations of Banker's algorithm have been applied in cloud computing where different computing tasks may be assigned to different platforms. Because resources available are usually limited on a given platform, it becomes necessary to check whether the tasks to be executed are schedulable.

We describe a variation of the resource allocation algorithm by a MSVL program where it is assumed that five tasks need to be scheduled every time and there exist four types of resources to be allocated. An array *maxres* is

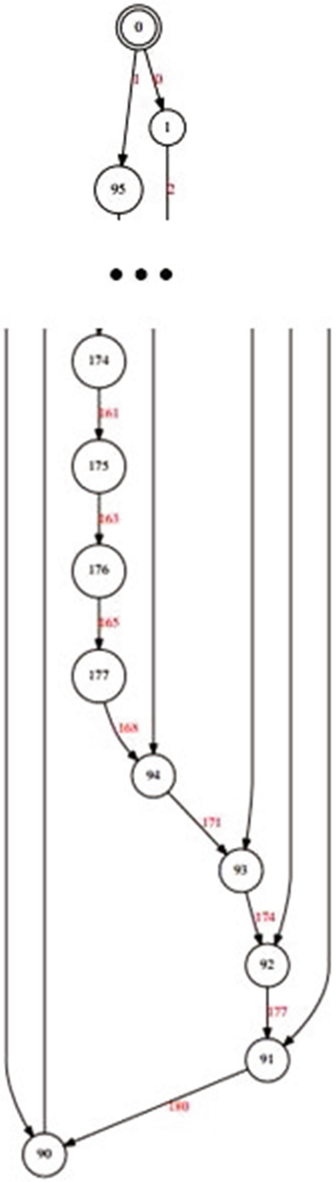


Fig. 5. The verification result of the resource allocation algorithm

used to indicate the number of resources available of each type. A 5 × 4 matrix *maxclaim* defines the maximum demand of each task and another 5 × 4 matrix *curr* defines the number of resources which are already allocated to each task. In the program, the platform keeps running through a nonterminal loop and the maximum demand of each task changes among three cases randomly. A boolean variable $fail$ is used to represent whether the tasks are schedulable in each case and $order$ is an array used to record every task's execution order.

Assume that we want to check whether or not the tasks are schedulable in each case and the third task is always executed after the second one. The property can be specified by $\Box(p \wedge q)$ in PPTL where p is defined as $fail = 0$ and q is defined as $order[1] < order[2]$. The upper bounded length is set to 100. Then we can verify the model with the bounded model checker. The verification result is shown in Fig. 5.

The counterexample is found when the bound increases to 89. By analysing the counterexample, we find that the resources available are too limited to schedule these five tasks. When the variable *maxclaim*=[[4, 2, 1, 4], [2, 2, 5, 2], [5, 1, 3, 5], [3 ,5 ,3 ,0], [3 ,2 ,3 ,3]], the first and the fourth types of resources are not enough. The program is verified again after we add one resource to the first and the fourth types respectively. Then we can get the result that the given property is valid for the modified program.

6 Conclusion

In this paper, we have proposed an approach named UBMC, which combines bounded model checking with unified model checking for verifying infinite state programs in MSVL. In our approach, a BLNFG is constructed on the fly to find whether there exist counterexamples in the given bound. This new technique produces counterexamples of minimal length and speeds up the verification. We also use a resource allocation example to show our approach. To examine our method, several case studies with larger examples are required in the near future. Moreover, lots of efforts are needed to improve our bounded model checker.

References

1. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logics of Programs. LNCS, pp. 52–71. Springer, Heidelberg (1982)
2. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
4. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. (TOPLAS) **16**(5), 1512–1542 (1994)
5. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: Proceedings of Fourth Annual Symposium on Logic in Computer Science, LICS 1989, IEEE, pp. 353–362 (1989)
6. McMillan, K.L.: Symbolic Model Checking. Springer, New York (1993)
7. Coudert, O., Madre, J.C.: A unified framework for the formal verification of sequential circuits. In: 1990 IEEE International Conference on Computer-Aided Design, ICCAD-90, Digest of Technical Papers, pp. 126–129. IEEE (1990)

8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
9. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
10. http://www.cs.cmu.edu
11. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., Vardi, M.Y.: Benefits of bounded model checking at an industrial setting. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 436–453. Springer, Heidelberg (2001)
12. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
13. Duan, Z.: An extended interval temporal logic and a framing technique for temporal logic programming. Ph.D. thesis, University of Newcastle upon Tyne (1996)
14. Tian, C., Duan, Z.: Propositional projection temporal logic, b*ü*chi automata and ω-regular expressions. In: Agrawal, M., Du, D.-Z., Duan, Z., Li, A. (eds.) TAMC 2008. LNCS, vol. 4978, pp. 47–58. Springer, Heidelberg (2008)
15. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems—Safety. Springer, New York (1995)
16. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, Orna (ed.) CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997)
17. Dingel, J., Filkorn, T.: Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In: Proceedings of 7th International Conference Computer Aided Verification, pp. 54–69 (1995)
18. Graf, S.: Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. Distrib. Comput. **12**(2–3), 75–90 (1999)
19. Mouawad, A.E., Nishimura, N., Raman, V., Simjour, N., Suzuki, A.: On the Parameterized Complexity of Reconfiguration Problems. In: Gutin, G., Szeider, S. (eds.) IPEC 2013. LNCS, vol. 8246, pp. 281–294. Springer, Heidelberg (2013)
20. McMillan, K.L.: Verification of infinite state systems by compositional model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 219–237. Springer, Heidelberg (1999)
21. Jhala, R., McMillan, K.L.: Microarchitecture verification by compositional model checking. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 396–410. Springer, Heidelberg (2001)
22. Schüle, T., Schneider, K.: Bounded model checking of infinite state systems. Formal Methods Syst. Des. **30**(1), 51–81 (2007)
23. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2005)
24. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Sci. Comput. Program. **70**(1), 31–61 (2008)
25. 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)
26. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Informatica **45**(1), 43–78 (2008)
27. Duan, Z., Tian, C.: A practical decision procedure for propositional projection temporal logic with infinite models. Theor. Comput. Sci. **554**, 169–190 (2014)
28. Dijkstra, E.W.: Cooperating Sequential Processes. Springer, New York (2002)

An Over-Approximation Forward Analysis for Nested Timed Automata

Yunqing Wen[1], Guoqiang Li[1](✉), and Shoji Yuen[2]

[1] School of Software, Shanghai Jiao Tong University, Shanghai, China
{wyqwyq,li.g}@sjtu.edu.cn
[2] Graduate School of Information Science, Nagoya University, Nagoya, Japan
yuen@is.nagoya-u.ac.jp

Abstract. *Nested timed automata (NeTAs)*, proposed by Li et al. are a pushdown system whose stack symbols are *timed automata (TAs)*. With this formal models, we can model and analyze complex real-time frameworks with recursive context switches. The reachability problem of NeTAs is proved to be decidable, via encoding NeTAs to *dense timed pushdown automata (DTPDAs)*. This paper gives a forward algorithm for reachability problem of NeTAs, by dividing the problem into two phases and integrating these two corresponding results. One phase is the reachability checking for the stack contents (i.e. TAs) and another is the state reachability problem for the TAs nested in an NeTA. The algorithm neglects time accumulation during context switches and thus an over-approximation of the original problem. As the result, the algorithm gains soundness in the sense that there exists one corresponding timed trace in the NeTA when the approximation has a timed trace to the state in less time-complexity.

1 Introduction

Nested timed automata (NeTAs) are a pushdown system whose stack symbols are *Timed Automata (TAs)*. It has been proposed to analyze recursive behavior of components in real-time systems, such as interrupts or procedure calls. It either behaves as the top TA in the stack, or switches from one TA to another following three kinds of transitions: pushing a new TA, popping the current TA when terminates, or replacing the top TA of the stack. A technical contribution of NeTA is that it allows local clocks in a natural way where the clocks are not frozen in the stack. The safety property of NeTAs is decidable [1] in general, by encoding NeTAs to the *dense timed pushdown automata (DTPDAs)* [2].

This paper proposes a forward analysis algorithm for NeTAs, as an approximation of configuration reachability analysis. NeTA may push the current TA on the top of the stack to switch to another TA. As modeling the real-time program behavior, the forward analysis from the initial state is natural and useful in checking non-reachability to the states with unintented properties. For example, in modelling non-maskable interrupts, pushing TA may not be controlled by the NeTA itself. In this case, the reachability may hold much more than the

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 62–80, 2015.
DOI: 10.1007/978-3-319-17404-4_5

general case since the push operation can happen at any time. Comparing with the general reachability checking, this may ease the algorithm in practice. The fundamental idea of the algorithm is to separate the timing behavior of TAs from the pushdown system. In this way, we can divide the reachability problem of NeTAs into two phases: reachability problem of TAs and reachability problem of *pushdown systems(PDSs)*, then we combine the results of two phases, as follows:

- The first phase can be judged by checking the traditional $\mathcal{P}$-automaton recognizing all the possible stack words(ignoring timing feature) for NeTA whether it satisfies some conditions.
- The second phase can be work out some by mature analysis algorithm such as zone construction to define a finite equivalence class on dense time.

The algorithm neglects time accumulation during context switches and thus just an over-approximation of the original problem. However, we have proved that the algorithm gains soundness by means that when a trace is found by the algorithm, there exists one corresponding timed trace in reality.

The rest of the paper is organized as follows. Section 2 gives an introduction of TAs, PDSs and NeTAs. Section 3 shows the formal definition of reachability problem of NeTAs and the forward analysis algorithm for NeTA. Section 4 proves the correctness of the forward analysis algorithm for NeTAs. The related work is presented in Sects. 5 and 6 concludes the paper.

2 Preliminaries

Let $\mathbb{R}^{\geq 0}$ and $\mathbb{N}$ denote the sets of non-negative real numbers and natural numbers respectively. Let $\mathbb{N}^{\omega} = \mathbb{N} \cup \{\omega\}$, where ω is the first limit ordinal. Let $\mathcal{I}$ denote the set of *intervals* over $\mathbb{N}^{\omega}$. An interval can be written as a pair of a lower limit and an upper limit in the form of either (a, b), $[a, b)$, $[a, c]$, $(a, c]$, where $a, c \in \mathbb{N}$, $b \in \mathbb{N}^{\omega}$,'(' and ')' denote open limits, and '[' and ']' denote closed limits. For a number $r \in \mathbb{R}^{\geq 0}$ and an interval $I \in \mathcal{I}$, we use $r \in I$ to denote that r belongs to I.

Let $X = \{x_1, \ldots, x_n\}$ be a finite set of *clocks*. A *clock valuation* $\nu : X \to \mathbb{R}^{\geq 0}$, assigns a value to each clock $x \in X$. $\nu_{\mathbf{0}}$ represents all clocks in X assigned to zero. Given a clock valuation ν and a time $t \in \mathbb{R}^{\geq 0}$, $(\nu + t)(x) = \nu(x) + t$, for $x \in X$. A clock assignment function $\nu[y \leftarrow b]$ is defined by $\nu[y \leftarrow b](x) = b$ if $x = y$, and $\nu(x)$ otherwise.

2.1 Timed Automata

Definition 1 (Timed Automata). *A timed automaton is a tuple* $\mathcal{A} = (Q, q_0, F, X, \Delta)$*, where*

- *Q is a finite set of control locations, with the initial location $q_0 \in Q$,*
- *$F \subseteq Q$ is the set of final locations,*

- *X is a finite set of clocks,*
- *$\Delta \subseteq Q \times \mathcal{O} \times Q$, where $\mathcal{O}$ is a set of* operations. *A transition $(q_1, \phi, q_2) \in \Delta$ is written as $q_1 \xrightarrow{\phi} q_2$, in which ϕ is either*
  Local *ϵ, an* empty *operation,*
  Test *$x \in I$? where $x \in X$ is a clock and $I \in \mathcal{I}$ is an interval, or*
  Assignment *$x \leftarrow I$ where $x \in X$ and $I \in \mathcal{I}$.*

Given a TA $\mathcal{A}$, we use $Q(\mathcal{A})$, $q_0(\mathcal{A})$, $F(\mathcal{A})$, $X(\mathcal{A})$ and $\Delta(\mathcal{A})$ to represent its set of control locations, initial location, set of final locations, set of clocks and set of transitions, respectively. We will use similar notations for other models.

Remark 1. We adopt the definition style of TAs from [1], which looks different from the one in [3,4]. The main reason lies in that this definition style is compatible with the definition of NeTAs [1]. Note that TAs defined in Definition 1 are diagonal-free. All test transitions only compare some clock with some constant, which can lead to the fact that all clock constraints have the form $x \sim c$, where $c \in \mathbb{N}$ and $\sim \in \{<, \leq, =, >, \geq\}$.

Definition 2 (Semantics of TAs). *Given a TA $\mathcal{A} = (Q, q_0, F, X, \Delta)$, a configuration(state) is a pair (q, ν) of a control location $q \in Q$, and a clock valuation ν on X. The transition relation of the TA is represented as follows:*

- Progress transition: *$(q, \nu) \xrightarrow{t}_{\mathscr{A}} (q, \nu + t)$, where $t \in \mathbb{R}^{\geq 0}$.*
- Discrete transition: *$(q_1, \nu_1) \xrightarrow{\phi}_{\mathscr{A}} (q_2, \nu_2)$, if $q_1 \xrightarrow{\phi} q_2 \in \Delta$, and one of the following holds:*
  - **Local** *$\phi = \epsilon$, then $\nu_1 = \nu_2$.*
  - **Test** *$\phi = x \in I$?, $\nu_1 = \nu_2$ and $\nu_2(x) \in I$ holds.*
  - **Assignment** *$\phi = x \leftarrow I$, $\nu_2 = \nu_1[x \leftarrow r]$ where $r \in I$.*

The initial configuration is $(q_0, \nu_{\mathbf{0}})$. The transition relation is $\rightarrow$ and we define $\rightarrow = \xrightarrow{t}_{\mathscr{A}} \cup \xrightarrow{\phi}_{\mathscr{A}}$, and define $\rightarrow^*$ to be the reflexive and transitive closure of $\rightarrow$.

Definition 3 (State Reachability Problem of TAs). *Given a timed automaton $\mathcal{A} = (Q, q_0, F, X, \Delta) \in \mathscr{A}$ and two locations p and q, where $p, q \in Q(\mathcal{A})$, determine whether there exist two clock valuation ν_1 and ν_2 such that $(q_0, \nu_{\mathbf{0}}) \rightarrow^* (p, \nu_1) \rightarrow^* (q, \nu_2)$. We write $(\mathcal{A}, p) \rightarrow^* (\mathcal{A}, q)$, if there exist such two clock valuations; otherwise $(\mathcal{A}, p) \not\rightarrow^* (\mathcal{A}, q)$.*

Remark 2. The definition style is different from the classical definition style [3]. The major difference is that we specify two locations p and q that must be reached in order from the initial control location q_0, while the classical definition requires at least one of final locations can be the reached from initial control location q_0. These two definitions are same in nature, since a modification of the algorithm for state reachability analysis of TAs goes for our definition.

The state reachability problem of TAs is equivalent to the *emptyness problem* of TAs. Some symbolical on-the-fly algorithms [5,6] based on the notion of zones are proposed for solving this problem, e.g. forward analysis algorithm used in Uppaal. For completeness, we present an algorithm for state reachability problem of TAs in Appendix A.

2.2 Pushdown System

Definition 4 (Pushdown Systems). *A pushdown system* $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ *is a quadruple where* P *contains the control locations and* Γ *is the stack alphabet. A configuration of* $\mathcal{P}$ *is a pair* $\langle p, w\rangle$ *where* $p \in P$ *and* $w \in \Gamma^*$*. Let* $Conf(\mathcal{P})$ *denote the set of all possible configurations.* c_0 *is the initial configuration. We associate a unique unlabelled transition system* $\mathcal{T}_{\mathcal{P}} = (Conf(P), \Rightarrow_{\mathcal{P}}, c_0)$ *with* $\mathcal{P}$*.*

Δ *is a finite subset of* $(P \times \Gamma) \times (P \times \Gamma^*)$*;we also write* $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ *if* $((p, \gamma), (p', w)) \in \Delta$*. The transition relation of* $\mathcal{T}_{\mathcal{P}}$ *is determined by the set of rules* Δ *as follows:*

$$IF\,\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle, \text{then}\,\langle p, \gamma w'\rangle \Rightarrow_{\mathcal{P}} \langle p', ww'\rangle\, forall\, w' \in \Gamma^*.$$

We define $\Rightarrow^*_{\mathcal{P}}$ *to be the reflexive and transitive closure of* $\Rightarrow_{\mathcal{P}}$*.*

Definition 5 (Forward Reachability Problem of PDSs). *Given a pushdown system* $\mathcal{P} = (P, \Gamma, \Delta, c_0)$*, and a target configuration* c_t*, determine whether there exists* $c_0 \Rightarrow^*_{\mathcal{P}} c_t$*.*

As for the forward reachability problem of PDSs, an well-known efficient algorithms [7] have been proposed based on the construction of $\mathcal{P}$-automaton, which can recognize all the configurations that can be reached from the initial configuration c_0 as a regular language.

Definition 6 ($\mathcal{P}$-automaton). *Given a pushdown system* $\mathcal{P} = (P, \Gamma, \Delta, c_0)$*, a* $\mathcal{P}$*-automaton is a quintuple* $\mathrm{A} = (Q, \Gamma, \rightarrow, P, F)$ *where* $Q \supseteq P$ *is a finite set of states,* $\rightarrow \subseteq Q \times \Gamma \times Q$ *is the set of transitions, and* $F \subseteq Q$ *the set of final states.* $\mathcal{P}$ *accepts or recognizes a configuration* $\langle p, w\rangle$ *if the transition system* $(Q, \Gamma, \rightarrow, p)$ *satisfies* $p \xrightarrow{w}{}^* q$ *for some* $q \in F$*.*

$\mathcal{P}$-automata are classified into *Post**-automata and *Pre**-automata. *Post**-automaton accepts the set of configurations that can be reached from initial configuration while *Pre**-automaton accepts the set of configurations that can reach the initial configuration. In our forward analysis, a *Post**-automaton is constructed to further reveal useful information.

2.3 Nested Timed Automata

Definition 7 (Nested Timed automata). *A nested timed automaton is a triplet* $\mathcal{N} = (T, \mathcal{A}_0, \Delta)$*, where*

- T *is a finite set of timed automata, with the initial timed automaton* $\mathcal{A}_0 \in T$*.*
- $\Delta \subseteq T \times \mathcal{P} \times (T \cup \{\varepsilon\})$*, where* $\mathcal{P} = \{push, pop, internal\}$*. A rule* $(\mathcal{A}_i, \Phi, \mathcal{A}_j) \in \Delta$ *is written as* $\mathcal{A}_i \xrightarrow{\Phi} \mathcal{A}_j$*, where*

  Push $\mathcal{A}_i \xrightarrow{push} \mathcal{A}_j$*,*

  Pop $\mathcal{A}_i \xrightarrow{pop} \varepsilon$*, and*

  Internal $\mathcal{A}_i \xrightarrow{internal} \mathcal{A}_j$*.*

The initial state of NeTAs is the initial location in $\mathcal{A}_0$, s.t. $q_0(\mathcal{A}_0)$. We also assume that $X(\mathcal{A}_i) \cap X(\mathcal{A}_j) = \emptyset$, and $Q(\mathcal{A}_i) \cap Q(\mathcal{A}_j) = \emptyset$ for $\mathcal{A}_i, \mathcal{A}_j \in T$ and $i \neq j$.

Definition 8 (Semantics of NeTAs). *Given a NeTA* $(T, \mathcal{A}_0, \Delta)$*, a configuration is a stack, and the stack alphabet is a tuple* $\langle \mathcal{A}, q, \nu \rangle$*, where* $\mathcal{A} \in T$ *is a timed automaton,* q *is the current running control location where* $q \in Q(\mathcal{A})$*, and* ν *is the clock valuation of* $X(\mathcal{A})$*. For a stack content* $c = \langle \mathcal{A}_1, q_1, \nu_1 \rangle \langle \mathcal{A}_2, q_2, \nu_2 \rangle \dots \langle \mathcal{A}_n, q_n, \nu_n \rangle$*, let* $c + t$ *be* $\langle \mathcal{A}_1, q_1, \nu_1 + t \rangle \langle \mathcal{A}_2, q_2, \nu_2 + t \rangle \dots \langle \mathcal{A}_n, q_n, \nu_n + t \rangle$*.*

The transition of NeTAs is represented as follows:

- *Progress transitions:* $c \xrightarrow{t}_{\mathcal{N}} c + t$*.*
- *Discrete transitions:* $c \xrightarrow{\phi}_{\mathcal{N}} c'$ *is defined as a union of the following four kinds of transition relations.*
  - **Intra-action** $\langle \mathcal{A}, q, \nu \rangle c \xrightarrow{\phi}_{\mathcal{N}} \langle \mathcal{A}, q', \nu' \rangle c$*, if* $q \xrightarrow{\phi} q' \in \Delta(\mathcal{A})$*, and one of the following holds:*
    - ∗ **Local** $\phi = \epsilon$*, then* $\nu = \nu'$*.*
    - ∗ **Test** $\phi = x \in I?$*,* $\nu = \nu'$ *and* $\nu'(x) \in I$ *holds.*
    - ∗ **Assignment** $\phi = x \leftarrow I$*,* $\nu' = \nu[x \leftarrow r]$ *where* $r \in I$*.*
  - **Push** $\langle \mathcal{A}, q, \nu \rangle c \xrightarrow{push}_{\mathcal{N}} \langle \mathcal{A}', q_0(\mathcal{A}'), \nu'_{\mathbf{0}} \rangle \langle \mathcal{A}, q, \nu \rangle c$*, if* $\mathcal{A} \xrightarrow{push} \mathcal{A}'$*, and* $q \in Q(\mathcal{A})$*.*
  - **Pop** $\langle \mathcal{A}, q, \nu \rangle c \xrightarrow{pop}_{\mathcal{N}} c$*, if* $\mathcal{A} \xrightarrow{pop} \varepsilon$*, and* $q \in F(\mathcal{A})$*.*
  - **Inter-action** $\langle \mathcal{A}, q, \nu \rangle c \xrightarrow{internal}_{\mathcal{N}} \langle \mathcal{A}', q_0(\mathcal{A}'), \nu'_{\mathbf{0}} \rangle c$*, if* $\mathcal{A} \xrightarrow{internal} \mathcal{A}'$*, and* $q \in F(\mathcal{A})$*.*

The initial configuration $c_0 = \langle \mathcal{A}_0, q_0(\mathcal{A}_0), \nu_{\mathbf{0}} \rangle$*. We use* $\longrightarrow$ *to range over the transitions above and* $\longrightarrow^*$ *is the reflexive and transitive closure of* $\longrightarrow$*, conventionally. We use* $\langle \mathcal{A}, q \rangle$ *to represent topmost TA in stack and its corresponding location.*

3 Forward Analysis for NeTAs

In this section, we first give the formal definition of reachability problem of *NeTAs*. Then, in order to concisely describe our forward analysis, we introduce how to construct a pushdown system from an NeTA and two important concepts: forward reachability and bi-directional reachability. The forward analysis algorithm is given at last.

3.1 Reachability Problem of NeTAs

Definition 9. *Given an NeTA* $\mathcal{N} = (T, \mathcal{A}_0, \Delta)$ *and two pairs* $\langle \mathcal{A}_i, p \rangle$*,* $\langle \mathcal{A}_j, q \rangle$*, where* $\mathcal{A}_i, \mathcal{A}_j \in T$*,* $p \in Q(\mathcal{A}_i)$*,*$q \in Q(\mathcal{A}_j)$*, determine whether there exist two configurations* c *and* c'*, such that* $c_0 = \langle \mathcal{A}_0, q_0(\mathcal{A}_0), \nu_{\mathbf{0}} \rangle \longrightarrow^* c = \langle \mathcal{A}_i, p, \nu \rangle \cdots \longrightarrow^* c' = \langle \mathcal{A}_j, q, \nu' \rangle \cdots$*, where* ν, ν' *are clock valuations over* $X(\mathcal{A}_i)$ *and* $X(\mathcal{A}_j)$ *respectively. We also write* $\langle \mathcal{A}_i, p \rangle \mapsto \langle \mathcal{A}_j, q \rangle$*, if* $\langle \mathcal{A}_i, p \rangle$ *can reach* $\langle \mathcal{A}_j, q \rangle$*.*

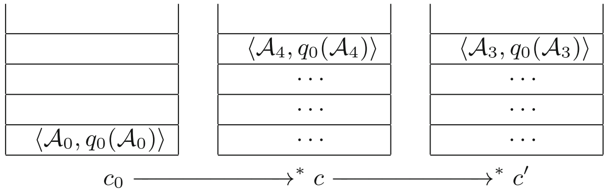
Consider a set of interrupts embedded in a real-time system. Some interrupts are unmasked and could happen any time. Thus wherever one interrupt is invoking its *interrupt handler*, another unmaskable interrupt could preempt the current executing interrupt and invoke its own *interrupt handler*. If the interrupt handlers share some common resources, it may be necessary to check their reachability. Assume two handlers happen to initialize some device. If both handlers may think they are the only handlers use the device, there might cause a problem: handler A wants to read data after the first initialization by itself. But if the initialization of B is reachable from the initialization of A, B might initialize the device unexpectedly. It actually requires to check two specific interrupts' reachability, which can be reduced to the reachability problem of NeTAs.

Example 1. An NeTA $\mathcal{N} = (T, \mathcal{A}_0, \Delta)$ is defined as follows:

- $\mathcal{T} = \{\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4\}$, where we assume $\mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4$ may reach one of their final locations respectively, while $\mathcal{A}_0, \mathcal{A}_1$ may not.
- $\mathcal{A}_0 \in T$ is the initial TA of $\mathcal{N}$;
- $\Delta = \{\mathcal{A}_0 \xrightarrow{push} \mathcal{A}_1, \mathcal{A}_1 \xrightarrow{pop} \varepsilon, \mathcal{A}_1 \xrightarrow{push} \mathcal{A}_2, \mathcal{A}_2 \xrightarrow{pop} \varepsilon, \mathcal{A}_2 \xrightarrow{internal} \mathcal{A}_3, \mathcal{A}_3 \xrightarrow{pop} \varepsilon, \mathcal{A}_1 \xrightarrow{push} \mathcal{A}_4, \mathcal{A}_4 \xrightarrow{pop} \varepsilon\}$.

Given $\mathcal{N}$ and two pairs $\langle \mathcal{A}_4, q_0(\mathcal{A}_4)\rangle$ and $\langle \mathcal{A}_3, q_0(\mathcal{A}_3)\rangle$, determine if $\langle \mathcal{A}_4, q_0(\mathcal{A}_4)\rangle \mapsto \langle \mathcal{A}_3, q_0(\mathcal{A}_3)\rangle$.

In Example 1, we need to determine if $\langle \mathcal{A}_4, q_0(\mathcal{A}_4)\rangle \mapsto \langle \mathcal{A}_3, q_0(\mathcal{A}_3)\rangle$. The intuition is that only when the two pairs could appear at the top of stack and one configuration with former pair $\langle \mathcal{A}_4, q_0(\mathcal{A}_4)\rangle$ at the top of stack could reach another configuration with latter pair $\langle \mathcal{A}_3, q_0(\mathcal{A}_3)\rangle$ at the top of stack, could $\langle \mathcal{A}_4, q_0(\mathcal{A}_4)\rangle \mapsto \langle \mathcal{A}_3, q_0(\mathcal{A}_3)\rangle$ be possible. The following figure shows the idea (Fig. 1).



$\langle \mathcal{A}_0, q_0(\mathcal{A}_0)\rangle$
$\langle \mathcal{A}_4, q_0(\mathcal{A}_4)\rangle$
$\cdots$
$\cdots$
$\cdots$
$\langle \mathcal{A}_3, q_0(\mathcal{A}_3)\rangle$
$\cdots$
$\cdots$
$\cdots$
$c_0 \longrightarrow^* c \longrightarrow^* c'$

Fig. 1. Reachable path in Example 1

3.2 PDS Contruction from a NeTA

Definition 10. *Given nested timed automata* $\mathcal{N} = (T, \mathcal{A}_0, \Delta)$*, we construct a pushdown system* $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ *from* $\mathcal{N}$*, where* $P = \{\bullet\}$*,* $\Gamma = \{\mathcal{A} | \mathcal{A} \in T(\mathcal{N})\}$*,* $c_0 = \langle \bullet, \mathcal{A}_0\rangle$*,*$\Delta \in (P \times \Gamma) \times (P \times \Gamma^{\leq 2})$ *defined as follows:*

- *if* $\mathcal{A}_i \xrightarrow{internal} \mathcal{A}_j \in \Delta(\mathcal{N})$ *for any* $\mathcal{A}_i \in T(\mathcal{N})$ *and* $\exists q_f \in F(\mathcal{A}_i)$ *s.t.* $(\mathcal{A}_i, q_0(\mathcal{A}_i)) \rightarrow^* (\mathcal{A}_i, q_f)$*, then* $\langle \bullet, \mathcal{A}_i \rangle \hookrightarrow \langle \bullet, \mathcal{A}_j \rangle$*;*
- *if* $\mathcal{A}_i \xrightarrow{push} \mathcal{A}_j \in \Delta(\mathcal{N})$ *for any* $\mathcal{A}_i \in T(\mathcal{N})$ *then* $\langle \bullet, \mathcal{A}_i \rangle \hookrightarrow \langle \bullet, \mathcal{A}_j \mathcal{A}_i \rangle$*;*
- *if* $\mathcal{A}_i \xrightarrow{pop} \varepsilon \in \Delta(\mathcal{N})$ *for any* $\mathcal{A}_i \in T(\mathcal{N})$ *and* $\exists q_f \in F(\mathcal{A}_i)$ *s.t.* $(\mathcal{A}_i, q_0(\mathcal{A}_i)) \rightarrow^* (\mathcal{A}_i, q_f)$ *, then* $\langle \bullet, \mathcal{A}_i \rangle \hookrightarrow \langle \bullet, \varepsilon \rangle$*.*

An algorithm for constructing $\mathcal{P}$-automaton from a PDS is given in Appendix B.

From the pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$, we can construct a $\mathcal{P}$-automaton $\mathcal{A}_{post*} = (Q, \Gamma, \rightarrow, P, F)$ which recognizes all possible configurations that are reachable from initial configuration c_0. Notice the fact that the $\mathcal{P}$-automaton $\mathcal{A}_{post*}$ has only one initial state whose labeled is $\bullet$. Since the initial state is frequently used in the following definitions and algorithms, let s_0 denote the distinctive initial state for simplicity.

3.3 Forward Reachability

Definition 11. *Given a* $\mathcal{P}$*-automaton* $\mathcal{A}_{post*} = (Q, \Gamma, \rightarrow, P, F)$ *which can recognize all possible reachable configurations of pushdown system* $\mathcal{P} = (P, \Gamma, \Delta, c_0)$*. We define* $\mathcal{A}_i \rightsquigarrow \mathcal{A}_j$*, where* $\mathcal{A}_i, \mathcal{A}_j \in \Gamma$ *if and only if there exists a transition upon* $\mathcal{A}_{post*}$ *in the form of* $s_0 \xrightarrow{\mathcal{A}_{k_0} = \mathcal{A}_j} s_1 \xrightarrow{\mathcal{A}_{k_1}} s_2 \cdots s_{n-1} \xrightarrow{\mathcal{A}_{k_{n-1}} = \mathcal{A}_i} s_n$*, where* $\mathcal{A}_{k_i} \in \Gamma$ *for any* k_i *and* $s_i \in Q(\mathcal{A}_{post*})$ *for* $0 \leq i \leq n$*.*

The intuition of the forward reachability of two TAs $\mathcal{A}_i \rightsquigarrow \mathcal{A}_j$ is the fact that when the topmost stack symbol is $\mathcal{A}_i$, it would be likely to use only "push" and "internal" transition rules to make the topmost symbol be $\mathcal{A}_j$ with $\mathcal{A}_i$ in stack.

Our forward analysis only cares about those TAs who can forward reach $\mathcal{A}_j$. Let $Fwd\_TA\_Set_{\mathcal{A}_j} = \{\mathcal{A}_k | \mathcal{A}_k \rightsquigarrow \mathcal{A}_j\}$. The following algorithm shows how to compute $Fwd\_TA\_Set_{\mathcal{A}_j}$.

Note in Algorithm 1, we assume there exists some $p \in Q(\mathcal{A}_{post*})$ such that $(s_0, \mathcal{A}_j, p) \in \rightarrow (\mathcal{A}_{post*})$. The basic idea of Algorithm 1 is to use the *breadth first search* method to search all transitions(or edges) concerned. In order to avoid revisiting the self-loop transitions, whenever a new transition is added to *tempEdgeSet*, only when it has never already been visited before.

3.4 Bi-directional Reachability

Definition 12. *Given a* $\mathcal{P}$*-automaton* $\mathcal{A}_{post*} = (Q, \Gamma, \rightarrow, P, F)$ *for pushdown system* $\mathcal{P} = (P, \Gamma, \Delta, c_0)$*, We define* $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_j$*, where* $\mathcal{A}_i, \mathcal{A}_j \in \Gamma$ *if and only if there exists a transition by* $\mathcal{A}_{post*}$ *in the form of* $s_0 \xrightarrow{\mathcal{A}_{k_0} = \mathcal{A}_i} s_1 \xrightarrow{\mathcal{A}_{k_1}} s_2 \cdots s_{n-1} \xrightarrow{\mathcal{A}_{k_{n-1}}} s_n$*, where for* $0 \leq i \leq n$*,*$s_i \in Q(\mathcal{A}_{post*})$*, for* $0 \leq i \leq n-1$*,*$\mathcal{A}_{k_i} \in \Gamma$*,*$\mathcal{A}_j \in \{\mathcal{A}_{k_{n-1}}\} \cup \{\gamma | (s_0, \gamma, q_{\bullet, \mathcal{A}_{k_{n-1}}}) \in \rightarrow (\mathcal{A}_{post*})\}$*, for* $0 \leq i < n-1$*,* $(s_0, \varepsilon, q_{\bullet, \mathcal{A}_{k_i}}) \in \rightarrow (\mathcal{A}_{post*})$*, and if* $\mathcal{A}_{k_{n-1}} \neq \mathcal{A}_j$*, then* $(s_0, \varepsilon, q_{\bullet, \mathcal{A}_{k_{n-1}}}) \in \rightarrow (\mathcal{A}_{post*})$*.*

Algorithm 1. Compute $Fwd\_TA\_Set_{\mathcal{A}_j}$

```
input  : $\mathcal{P}$-automaton $\mathcal{A}_{post*} = (Q, \Gamma, \rightarrow, P, F)$; TA $\mathcal{A}_j$ that need to be forward
         reached
output: the set of TAs $Fwd\_TA\_Set_{\mathcal{A}_j} = \{\mathcal{A}_k | \mathcal{A}_k \rightsquigarrow \mathcal{A}_j\}$
$Vistied := \emptyset, EdgeSet := \emptyset$;
forall the $(s_0, \mathcal{A}_j, p) \in\rightarrow (\mathcal{A}_{post*})$ do
    $EdgeSet := EdgeSet \cup \{(s_0, \mathcal{A}_j, p)\}$;
    $Visited := Visited \cup \{(s_0, \mathcal{A}_j, p)\}$;
do
    $tempEdgeSet := \emptyset$;
    forall the $(p, \gamma, q) \in EdgeSet$ do
        forall the $(q, \gamma', q') \in\rightarrow (\mathcal{A}_{post*})$ do
            if $(q, \gamma', q') \notin Visited$ then
                $Visited := Visited \cup \{(q, \gamma', q')\}$;
                $tempEdgeSet := tempEdgeSet \cup \{(q, \gamma', q')\}$;
    $EdgeSet := tempEdgeSet$;
while $EdgeSet \neq \emptyset$;
$Fwd\_TA\_Set_{\mathcal{A}_j} := \{\gamma | (p, \gamma, q) \in Visited\}$;
return $Fwd\_TA\_Set_{\mathcal{A}_j}$;
```

The intuition of the bi-directional reachability of $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_j$ is the fact that when the topmost stack symbol is $\mathcal{A}_i$, it would be likely to use only "pop" and "internal" transition rules to make the topmost symbol be $\mathcal{A}_j$ without $\mathcal{A}_i$ remained in stack. In Example 1, it's easy to find that $\mathcal{A}_2 \leftrightsquigarrow \mathcal{A}_1$, since when the topmost symbol $\mathcal{A}_2$ is popped out, the next topmost symbol could be $\mathcal{A}_1$. The specific situation is shown in the following figure.

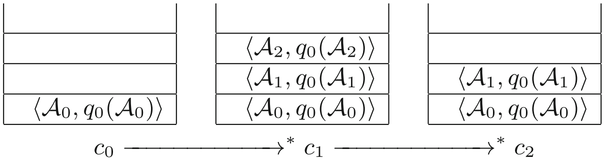

$\langle \mathcal{A}_0, q_0(\mathcal{A}_0) \rangle$
$\langle \mathcal{A}_2, q_0(\mathcal{A}_2) \rangle$
$\langle \mathcal{A}_1, q_0(\mathcal{A}_1) \rangle$
$\langle \mathcal{A}_0, q_0(\mathcal{A}_0) \rangle$
$\langle \mathcal{A}_1, q_0(\mathcal{A}_1) \rangle$
$\langle \mathcal{A}_0, q_0(\mathcal{A}_0) \rangle$
$c_0 \longrightarrow^* c_1 \longrightarrow^* c_2$

Fig. 2. A demonstration of $\mathcal{A}_2 \leftrightsquigarrow \mathcal{A}_1$ in Example 1

Algorithm 2 shows how to compute those TAs that can be bi-directionally reached from $\mathcal{A}_i$.

Note in Algorithm 2, we also assume there exists some $p \in Q(\mathcal{A}_{post*})$ such that $(s_0, \mathcal{A}_i, p) \in\rightarrow (\mathcal{A}_{post*})$. The basic idea underlying Algorithm 2 is similar to that of Algorithm 1. The major difference is to ensure all TAs along the bi-directional path except the last one should be able to popped out of the stack.

Algorithm 2. Compute $Bi\_TA\_Set_{\mathcal{A}_i}$

```
input  : $\mathcal{P}$-automaton $\mathcal{A}_{post*} = (Q, \Gamma, \to, P, F)$; TA $\mathcal{A}_i$ that bi-directionally
         reach other TAs
output: the set of TAs $Bi\_TA\_Set_{\mathcal{A}_i} = \{\mathcal{A}_k | \mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k\}$
$Visited := \emptyset, EdgeSet := \emptyset$;
forall the $(s_0, \gamma, p) \in \to (\mathcal{A}_{post*})$ do
    if $\gamma$ is $\mathcal{A}_i$ or $p$ is $q_{\bullet,\mathcal{A}_i}$ then
        $EdgeSet := EdgeSet \cup \{(s_0, \gamma, p)\}$;
        $Visited := Visited \cup \{(s_0, \gamma, p)\}$;
do
    $tempEdgeSet := \emptyset$;
    forall the $(p, \gamma, q) \in EdgeSet$ do
        if $(s_0, \varepsilon, q_{\bullet,\gamma}) \in \to (\mathcal{A}_{post*})$ then
            forall the $(q, \gamma', q') \in \to (\mathcal{A}_{post*})$ do
                if $(q, \gamma', q') \notin Visited$ then
                    $Visited := Visited \cup \{(q, \gamma', q')\}$;
                    $tempEdgeSet := tempEdgeSet \cup \{(q, \gamma', q')\}$;
                    if $(s_0, \varepsilon, q_{\bullet,\gamma'}) \in \to (\mathcal{A}_{post*})$ then
                        forall the $(s_0, \gamma'', q_{\bullet,\gamma'}) \in \to (\mathcal{A}_{post*})$ do
                            if $(s_0, \gamma'', q_{\bullet,\gamma'}) \notin Visited$ then
                                $Visited := Visited \cup \{(s_0, \gamma'', q_{\bullet,\gamma'})\}$;
                                $tempEdgeSet := tempEdgeSet \cup \{(s_0, \gamma'', q_{\bullet,\gamma'})\}$;
    $EdgeSet := tempEdgeSet$;
while $EdgeSet \neq \emptyset$;
$Bi\_TA\_Set_{\mathcal{A}_i} := \{\gamma | (p, \gamma, q) \in Visited\}$;
return $Bi\_TA\_Set_{\mathcal{A}_i}$;
```

3.5 Forward Analysis Algorithm for NeTAs

Our forward analysis algorithm for NeTAs is presented in Algorithm 3.

The idea underlying it is quite simple: to check whether $\langle \mathcal{A}_j, q \rangle$ is reachable from $\langle \mathcal{A}_i, p \rangle$, firstly, locally check whether p and q is reachable from initial locations in $\mathcal{A}_i$ and $\mathcal{A}_j$ and whether reach final locations, respectively, by region construction, without considering context switch; secondly, check whether $\mathcal{A}_j$ is reachable from $\mathcal{A}_i$, by $\mathcal{P}$-automaton technique, without considering time elapse. In other words, check whether there exists a TA $\mathcal{A}_k \in T(\mathcal{N})$ such that $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k \rightsquigarrow \mathcal{A}_j$ hold; finally, combine the results of two stages. In our algorithm two cases are considered separately: One case is that $\mathcal{A}_i$ can directly forwardly reach $\mathcal{A}_j$(i.e. $\mathcal{A}_i \rightsquigarrow \mathcal{A}_j$) and the other case is that $\mathcal{A}_i$ can not directly forwardly reach $\mathcal{A}_j$ but there exists a $\mathcal{A}_k$ that act as a *bridge* connecting $\mathcal{A}_i$ and $\mathcal{A}_j$ (i.e. $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k \rightsquigarrow \mathcal{A}_j$). In the latter case, it is required that the $\mathcal{A}_i$ can reach one of its final locations from the specified location p by the definition of $\leftrightsquigarrow$.

Algorithm 3. Forward Analysis Algorithm for NeTAs

```
input  : NeTA $\mathcal{N} = (T, \mathcal{A}_0, \Delta)$; two pairs $\langle \mathcal{A}_i, p \rangle, \langle \mathcal{A}_j, q \rangle$, where $\mathcal{A}_i, \mathcal{A}_j \in T(\mathcal{N})$,
         $p \in Q(\mathcal{A}_i)$ and $q \in Q(\mathcal{A}_j)$
output: if $\langle \mathcal{A}_i, p \rangle \mapsto \langle \mathcal{A}_j, q \rangle$ output “Yes”; otherwise output “No”
1  if $(\mathcal{A}_i, q_0(\mathcal{A}_i)) \not\rightarrow^* (\mathcal{A}_i, p)$ then
2      return “No”;
3  if $(\mathcal{A}_j, q_0(\mathcal{A}_j)) \not\rightarrow^* (\mathcal{A}_j, q)$ then
4      return “No”;
5  With NeTA $\mathcal{N} = (T, \mathcal{A}_0, \Delta)$, construct a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$
   according to definition 10;
6  Construct a $\mathcal{P}$-automaton $\mathcal{A}_{post*} = (Q, \Gamma, \rightarrow, P, F)$ for PDS $\mathcal{P}$;
7  if $\nexists s\ s.t.(s_0, \mathcal{A}_i, s) \in \rightarrow (\mathcal{A}_{post*})$ or $\nexists s\ s.t.(s_0, \mathcal{A}_j, s) \in \rightarrow (\mathcal{A}_{post*})$ then
8      return “No”;
9  Use algorithm 1 to search TAs that can forward reach $\mathcal{A}_j$,
   $Fwd\_TA\_Set_{\mathcal{A}_j} = \{\mathcal{A}_k | \mathcal{A}_k \rightsquigarrow \mathcal{A}_j\}$;
10 if $\mathcal{A}_i \in Fwd\_TA\_Set_{\mathcal{A}_j}$ then
11     return “Yes”;
12 if $\nexists q_f \in F(\mathcal{A}_i)\ s.t.(\mathcal{A}_i, p) \rightarrow^* (\mathcal{A}_i, q_f)$ then
13     return “No”;
14 Use algorithm 2 to search TAs that can be bi-directionally reached from $\mathcal{A}_i$,
   $Bi\_TA\_Set_{\mathcal{A}_i} = \{\mathcal{A}_k | \mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k\}$;
15 if $Bi\_TA\_Set_{\mathcal{A}_i} \wedge Fwd\_TA\_Set_{\mathcal{A}_j} \neq \emptyset$ then
16     return “Yes”;
17 else
18     return “No”;
```

For better upstanding we describe our algorithm by applying it to example 1. The corresponding $\mathcal{P}$ automaton is shown in Fig. 3. Our algorithm in lines 1–4 first checks whether two TAs $\mathcal{A}_4, \mathcal{A}_3$ from their initial locations can reach their respective locations $q_0(\mathcal{A}_4), q_0(\mathcal{A}_3)$. If do not hold, it trivially return a “No”. Obviously, due to the reflexiveness of →, these 2 conditions hold. In lines 5–6, a $\mathcal{P}$-automaton $\mathcal{A}_{post*}$ is constructed which can recognize all possible stack configurations ignoring the internal change of TAs. The $\mathcal{P}$-automaton $\mathcal{A}_{post*}$ is showed in the Fig. 3. In lines 7–8, it checks whether both TAs can appear at the top of stack. If can not, trivially return a “No”. In Fig. 3, there are 2 transition rules $s_0 \xrightarrow{\mathcal{A}_4} q_{\bullet,\mathcal{A}_4}$ and $s_0 \xrightarrow{\mathcal{A}_3} q_{\bullet,\mathcal{A}_2}$, meaning both of 2 TAs can appear at the top of stack. Go on to line 9. Our algorithm find all TAs that each of them can forward reach $\mathcal{A}_3$ by using Algorithm 1. Let $Fwd\_TA\_Set_{\mathcal{A}_3}$ denote the set of TAs. From Fig. 3, we can find $Fwd\_TA\_Set_{\mathcal{A}_3} = \{\mathcal{A}_3, \mathcal{A}_1, \mathcal{A}_0\}$. In lines 10–11, it checks whether $\mathcal{A}_4 \in Fwd\_TA\_Set_{\mathcal{A}_3}$. If so, return “Yes”. Clearly, $\mathcal{A}_4 \notin Fwd\_TA\_Set_{\mathcal{A}_3}$. In lines 12–13, it checks whether $\mathcal{A}_4$ can reach some final locations from current location p. If not, return “No”. By assumption, $\mathcal{A}_4$ can reach some final locations. In line 14, we find all TAs that can be bi-directionally

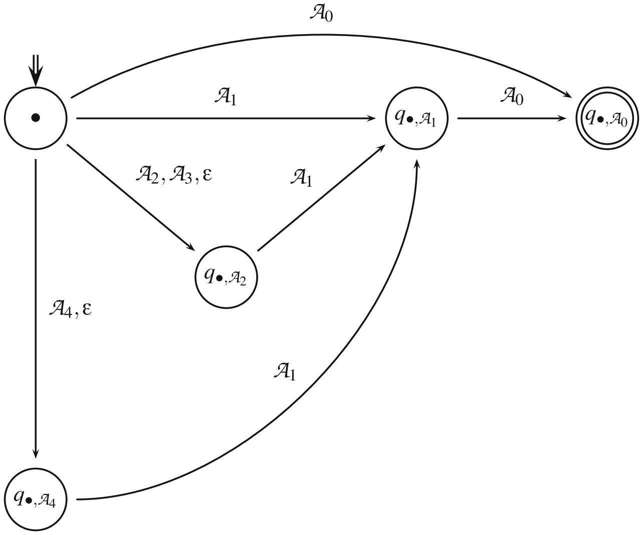

$\mathcal{A}_0$
$\mathcal{A}_1$
$\mathcal{A}_0$
$\mathcal{A}_2, \mathcal{A}_3, \varepsilon$
$\mathcal{A}_1$
$q_{\bullet,\mathcal{A}_1}$
$q_{\bullet,\mathcal{A}_0}$
$q_{\bullet,\mathcal{A}_2}$
$\mathcal{A}_4, \varepsilon$
$\mathcal{A}_1$
$q_{\bullet,\mathcal{A}_4}$

Fig. 3. $\mathcal{P}$-automaton constructed in Example 1

reached from $\mathcal{A}_4$ by using Algorithm 2. Let $Bi\_TA\_Set_{\mathcal{A}_4}$ denote the set of TAs. It's easy to find $Bi\_TA\_Set_{\mathcal{A}_4} = \{\mathcal{A}_4, \mathcal{A}_1\}$. In lines 15–18, it checks whether the intersection of two sets(i.e. TAs $Fwd\_TA\_Set_{\mathcal{A}_3}$ and $Bi\_TA\_Set_{\mathcal{A}_4}$) is empty or not. Since both sets have a common TA $\mathcal{A}_1$, our algorithm return a "Yes".

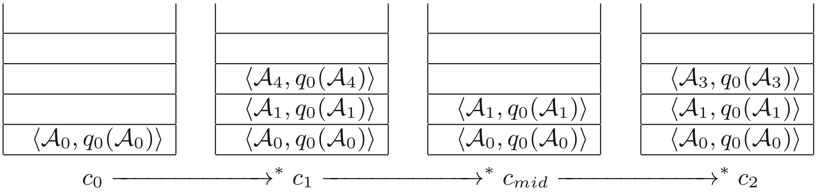

$\langle \mathcal{A}_0, q_0(\mathcal{A}_0) \rangle$
$\langle \mathcal{A}_4, q_0(\mathcal{A}_4) \rangle$
$\langle \mathcal{A}_1, q_0(\mathcal{A}_1) \rangle$
$\langle \mathcal{A}_0, q_0(\mathcal{A}_0) \rangle$
$\langle \mathcal{A}_1, q_0(\mathcal{A}_1) \rangle$
$\langle \mathcal{A}_0, q_0(\mathcal{A}_0) \rangle$
$\langle \mathcal{A}_3, q_0(\mathcal{A}_3) \rangle$
$\langle \mathcal{A}_1, q_0(\mathcal{A}_1) \rangle$
$\langle \mathcal{A}_0, q_0(\mathcal{A}_0) \rangle$
$c_0 \longrightarrow^* c_1 \longrightarrow^* c_{mid} \longrightarrow^* c_2$

Fig. 4. One possible reachable path in Example 1

In the computation process, we actually find a relatively coarse but not precise reachable path that witness $\langle \mathcal{A}_4, q_0(\mathcal{A}_4) \rangle \mapsto \langle \mathcal{A}_3, q_0(\mathcal{A}_3) \rangle$. Figure 4 shows one of the possible reachable paths.

Remark 3. Our algorithm is sound in the sense that it can find a relatively coarse reachable path and return "Yes" if and only if $\langle \mathcal{A}_i, p \rangle \mapsto \langle \mathcal{A}_j, q \rangle$. However, our algorithm is not precise as the reachable paths we find are not detailed enough to construct a concrete reachable path with timing trace. Actually, it neglects time accumulation during context switches. Besides, our algorithm only search some "best-case" reachable paths, not capable of systematically searching all possible path. We say these reachable paths are "best-case" in the sense that although it could have infinitely many concrete reachable paths, any concrete reachable path's backbone is one of these "best-case" reachable paths, sice a preemption and resumption can be simulated by a progress transition of a local TA. In terms of the above analysis our algorithm is over-approximation.

We find it difficult to design a precise algorithm that could search all reachable paths systematically. The reason lies in that there could be infinitely many reachable paths and each of them could differs greatly with others, implying unlikely to find some effective way to search all reachable paths. Despite of imprecision, our algorithm is capable for almost all cases since the safety property are mostly concerned and the precise timing information is usually not important.

4 Correctness of Forward Analysis Algorithm for NeTAs

In this section, we prove our algorithm's correctness.

Lemma 1. *Given an NeTA $\mathcal{N} = (T, \mathcal{A}_0, \Delta)$ and two pairs $\langle \mathcal{A}_i, p\rangle$, $\langle \mathcal{A}_j, q\rangle$, where $\mathcal{A}_i, \mathcal{A}_j \in T$, a unique $\mathcal{P}$-automaton $\mathcal{A}_{post*} = (Q, \Gamma, \rightarrow, P, F)$ is associated with NeTA $\mathcal{N}$. If $\langle \mathcal{A}_i, p\rangle \mapsto \langle \mathcal{A}_j, q\rangle$, the following 4 properties hold:*

1. *$\exists s_1, s_2 \in Q(\mathcal{A}_{post*})$ s.t. $(s_0, \mathcal{A}_i, s_1), (s_0, \mathcal{A}_j, s_2) \in \rightarrow (\mathcal{A}_{post*})$;*
2. *for TAs $\mathcal{A}_i$ and $\mathcal{A}_j$, the following hold:*
   - $(\mathcal{A}_i, q_0(\mathcal{A}_i)) \rightarrow^* (\mathcal{A}_i, p)$
   - $(\mathcal{A}_j, q_0(\mathcal{A}_j)) \rightarrow^* (\mathcal{A}_j, q)$
3. *$\exists \mathcal{A}_k \in T(\mathcal{N})$ s.t. $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k \rightsquigarrow \mathcal{A}_j$;*
4. *if $\mathcal{A}_i \rightsquigarrow \mathcal{A}_j$ doesn't hold, then $\exists q_f \in F(\mathcal{A}_i)$ s.t. $(\mathcal{A}_i, p) \rightarrow^* (\mathcal{A}_i, q_f)$.*

Lemma 2. *Given an NeTA $\mathcal{N} = (T, \mathcal{A}_0, \Delta)$ and two pairs $\langle \mathcal{A}_i, p\rangle$, $\langle \mathcal{A}_j, q\rangle$, where $\mathcal{A}_i, \mathcal{A}_j \in T$, a unique $\mathcal{P}$-automaton $\mathcal{A}_{post*} = (Q, \Gamma, \rightarrow, P, F)$ is associated with NeTA $\mathcal{N}$. If the following 4 properties hold:*

1. *$\exists s_1, s_2 \in Q(\mathcal{A}_{post*})$ s.t. $(s_0, \mathcal{A}_i, s_1), (s_0, \mathcal{A}_j, s_2) \in \rightarrow (\mathcal{A}_{post*})$;*
2. *for TAs $\mathcal{A}_i$ and $\mathcal{A}_j$, the following hold:*
   - $(\mathcal{A}_i, q_0(\mathcal{A}_i)) \rightarrow^* (\mathcal{A}_i, p)$
   - $(\mathcal{A}_j, q_0(\mathcal{A}_j)) \rightarrow^* (\mathcal{A}_j, q)$
3. *$\exists \mathcal{A}_k \in T(\mathcal{N})$ s.t. $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k \rightsquigarrow \mathcal{A}_j$;*
4. *if $\mathcal{A}_i \rightsquigarrow \mathcal{A}_j$ doesn't hold, then $\exists q_f \in F(\mathcal{A}_i)$ s.t. $(\mathcal{A}_i, p) \rightarrow^* (\mathcal{A}_i, q_f)$.*

then $\langle \mathcal{A}_i, p\rangle \mapsto \langle \mathcal{A}_j, q\rangle$.

The proofs of Lemmas 1 and 2 are given in Appendices C and D respectively.

Theorem 1 (Correctness Of Algorithm 3). *Algorithm 3 can correctly decide the reachability problem of NeTAs.*

Algorithm 3 determine the reachability problem of NeTAs by checking the 4 properties described in above 2 lemmas. With Lemma 1 and 2, Algorithm 3 can correctly decide the reachability problem of NeTAs.

5 Related Work

Reachability of *timed automata (TAs)* is proved to be decidable based on the construction of *region automata*, which finitely abstracts timed behaviors of TA [3]. However, this construction does not lead to a practical reachability algorithm of TA due to an enormous combinatorics explosion. Symbolic on-the-fly algorithms was proposed and implemented to avoid the complexity of blow-up caused by timing constraints. These algorithms were based on *zones* and *DBMs* to represent sets of clock valuations. For practical purpose, these zone-based algorithms could be used in our forward analysis.

In NeTA, only simple clock updates and diagonal-free time constraints are allowed. The *updatable timed automata (UTAs)* [8] was a natural syntactic extension of TA. It enjoyed the possibility of updating the clocks in a more elaborate way than just simple reset in TA. The undecidability and decidability results were given for several specific cases [8]. A forward analysis of UTA has been proposed in [9] for specific subclass of UTA that do not use comparisons between clocks.

Decidability of the reachability problem of NeTA is proved by translation to *Dense-timed pushdown automata (DTPDAs)* [2]. *DTPDAs* extend the classical models of pushdown automata and timed automata, in the sense that the automaton operates on a finite set of real-valued clocks, and each symbol in the stack is equipped with a real-valued clock. The problem of reachability has been proven to be decidable for *DTPDAs*. It relies on constructing a classical untimed pushdown automaton over time abstract. However, the untimed automaton produced generally contains a very large number of states. Although a zone-based reachability analysis of *DTPDAs* has been proposed in [10], the precise forward analysis may require a significant effort compared to our algorithm.

Recursive timed automata(RTA) [11] is an extension of timed automata with recursive structure. It has local clocks by the mechanism of "pass-by-value". When the condition of "glitch-freeness", i.e. all the clocks of components are uniformly either by "pass-by-value" or by "pass-by-reference", the reachability is shown to be decidable. RTAs have a mechanism of freezing clocks while in the stack by 'pass-by-value' while NeTA do not. Since our algorithm ignores the accumulation of time while being in the stack, a similar technique may be applied to RTAs for estimating rough reachability.

6 Conclusion

We gave a forward analysis algorithm for NeTA to check the state reachability. The basic idea of algorithm is to divide the reachability problem of NeTAs into two phases: one phase is the reachability checking for the stack contents and another is the state reachability problem for the TAs nested in an NeTA. Although the algorithm is over-approximation, it is sound and complete in the sense that when it can find an over-approximate trace ignoring the accumulation of time in the stack, there exists one such reachable timed trace and vice versa.

We proved the correctness of the algorithm in Sect. 4. The proof is based on the insight of 4 properties that are an sound approximation to the reachablity problem of NeTAs. Our algorithm complexity is **PSPACE-Complete** as the reachability algorithm of TAs is used, which is **PSPACE-Complete**, and the rest can be done in polynomial time and polynomial space. This shows that our algorithm may be used for verification of real-time systems with elaborate data structures such as *DBM*.

Our algorithm will not work for the NeTAs with *invarints*, in which each TAs' location is assigned to a clock constraint. The reason lies in that our algorithm neglects the time accumulation during context switches but it has to be considered since we need to ensure every discrete and progress transition would lead to a *valid* configuration. However, our algorithm could be applying *counter-example-guided-abstraction-refinement (CEGAR)* to NeTAs with invariants, which is our future work.

Acknowledgment. The authors thank Prof. Mizuhito Ogawa for fruitful discussions on the research. This research is supported by the National Nature Science Foundation of China (61100052, 61472240, 91318301, 61261130589), JSPS Kakenhi Grant-in-Aid for Scientific Research(B) (25280023) and Challenging Exploratory Research(26540026).

A An Algorithm for State Reachability Problem of TAs

An algorithm based on the notion of region is given in the following. In order to better describe the algorithm, we need to introduce some definitions first.

Let C be the maximal clock constant appearing in the TA. Given a clock valuation ν over a set of clock $X = \{x_1, \ldots, x_n\}$,and a time $t \in \mathbb{R}^{\geq 0}$, $(\nu + t)(x)$ is redefined by $\nu(x) + t$ if $\nu(x) + t \leq C$ and otherwise any non-integral value C' whose integral part is C. A function $isInt(x)$ is defined by 1 if $\nu(x)$ is an integer and 0 otherwise, determining whether the value of a given clock is integer or not. A vector $\boldsymbol{H}(\nu)$,which characterizes integral properties of a clock valuation ν, is defined by $(isInt(x_1), \ldots, isInt(x_n))$.

For any time interval $I \in \mathcal{I}$ that appears in the TA, we bound it in the following way: $I_b = \{r | r \in I \wedge r \leq C'\}$. Without confusion, any interval mentioned in the following is bounded.

Definition 13 (Region Equivalence). *For a real number d, let $\{d\}$ denote the fractional part of d, and $\lfloor d \rfloor$ denote its integer part. We say, two clock valuations ν_1, ν_2 are region-equivalent, denoted $\nu_1 \sim \nu_2$, if and only if*

1. *for all x, either $\lfloor \nu_1(x) \rfloor = \lfloor \nu_2(x) \rfloor$ or both $\nu_1(x) > C$ and $\nu_2(x) > C$,*
2. *for all x, if $\nu_1(x) \leq C$ then $\{\nu_1(x)\} = 0$ iff $\{\nu_2(x)\} = 0$ and,*
3. *for all x, y if $\nu_1(x) \leq C$ and $\nu_1(y) \leq C$ then $\{\nu_1(x)\} \leq \{\nu_1(y)\}$ iff $\{\nu_2(x)\} \leq \{\nu_2(y)\}$.*

The equivalence class $[\nu_1]$ induced by $\sim$ is called a region, where $[\nu_1]$ represent all clock valuations that is region-equivalent to ν_1.

A symbolic state of TA is a pair $\langle q, R\rangle$ representing a set of states of the TA, where p is a location and R is a region. A symbolic transition describes all the possible concrete transitions from the set of states.

Definition 14. *Given a region R, we define its direct successor: $R^{\uparrow} = \{\nu+t | \nu \in R \wedge \boldsymbol{H}(\nu+t) = \min_{t'} \boldsymbol{H}(\nu+t') \neq \boldsymbol{H}(\nu)\}$, where $t, t' \in \mathbb{R}^{\geq 0}$. We define $R(x) \models I$ if $\forall \nu \in R, \nu(x) \in I$, and $I_x(R) = \{[\nu[x \leftarrow r]] | \nu \in R \wedge r \in I\}$, where $x \in X$ is a clock and I is a time interval. Let $\rightsquigarrow$ denote the symbolic transition relation over symbolic states defined by the following rules:*

- $\langle q, R\rangle \rightsquigarrow \langle q, R^{\uparrow}\rangle$
- $\langle q, R\rangle \rightsquigarrow \langle q', R\rangle$, *if* $q \xrightarrow{\varepsilon} q'$
- $\langle q, R\rangle \rightsquigarrow \langle q', R\rangle$, *if* $q \xrightarrow{x \in I?} q'$ *and* $R \models I$
- $\forall R' \in I_x(R), \langle q, R\rangle \rightsquigarrow \langle q', R'\rangle$, *if* $q \xrightarrow{x \leftarrow I} q'$

With the above definitions, we give a reachability algorithm for diagonal-free TAs. In Algorithm 4, we use a boolean variable *flag* to indicate whether the corresponding location is reached from p or not.

Algorithm 4. An Algorithm for state reachability problem of TAs

```
input : TA $\mathcal{A} = (Q, q_0, F, X, \Delta)$; two control locations $p, q \in Q(\mathcal{A})$
output: "yes" if $(\mathcal{A}, p) \rightarrow^* (\mathcal{A}, q)$;"no" otherwise
if $p = q_0(\mathcal{A})$ then
    $flag := true$;
else
    $flag := false$;
$PASSED := \emptyset, WAIT := \{\langle q_0(\mathcal{A}), \{\nu_0\}, flag\rangle\}$;
while $WAIT \neq \emptyset$ do
    take $\langle p', R, flag\rangle$ from $WAIT$;
    if $p' = q \wedge flag = true$ then
        return "yes";
    if $p' = p \wedge flag = false$ then
        $flag := true$;
    if $\langle p', R, flag\rangle \notin PASSED$ then
        $PASSED := PASSED \cup \{\langle p', R, flag\rangle\}$;
        forall the $\langle q', R'\rangle$ such that $\langle p', R\rangle \rightsquigarrow \langle q', R'\rangle$ do
            add $\langle q', R', flag\rangle$ to $WAIT$;
return "no";
```

Remark 4. Algorithm 4 is based on *region*, which could be computational expensive. Although it could be optimized by using the well-known *zone* technique, it does not improve its theoretical complexity.

B An Algorithm for $\mathcal{P}$-Automaton from PDS

The following is an effective algorithm for constructing $\mathcal{P}$-automaton from a PDS, which is taken from [7].

Algorithm 5. An algorithm for constructing a $\mathcal{P}$-automaton from PDS

```
input  : a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$;
         a $\mathcal{P}$-Automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$ without transitions into $P$ and
         without $\varepsilon$-transitions
output: the automaton $\mathcal{A}_{post*}$
$trans := (\rightarrow_0) \cap (P \times \Gamma \times Q)$;
$rel := (\rightarrow_0) \backslash trans, Q' := Q$;
forall the $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1\gamma_2 \rangle \in \Delta$ do
    $Q' := Q' \cup \{q_{p',\gamma_1}\}$;
while $trans \neq \emptyset$ do
    pop $t = (p, \gamma, q)$ from $trans$;
    if $t \notin rel$ then
        $rel := rel \cup \{t\}$;
        if $\gamma \neq \varepsilon$ then
            forall the $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$ do
                $trans := trans \cup \{(p', \varepsilon, q)\}$;
            forall the $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1 \rangle \in \Delta$ do
                $trans := trans \cup \{(p', \gamma_1, q)\}$;
            forall the $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1\gamma_2 \rangle \in \Delta$ do
                $trans := trans \cup \{(p', \gamma_1, q_{p',\gamma_1})\}$;
                $rel := rel \cup \{(q_{p',\gamma_1}, \gamma_2, q)\}$;
                forall the $(p'', \varepsilon, q_{p',\gamma_1}) \in rel$ do
                    $trans := trans \cup \{(p'', \gamma_2, q)\}$;
        else
            forall the $(q, \gamma', q') \in rel$ do
                $trans := trans \cup \{(q, \gamma', q')\}$;
return $(Q', \Gamma, rel, P, F)$;
```

With out of generality,we assume that A has no transition leading to an initial state. For Algorithm 5, the input is a pushdown system $\mathcal{P} = (P, \Gamma, \Delta, c_0)$ and an automaton $\mathcal{A}$ accepting c_0, and the output is an automaton $\mathcal{A}_{post*}$ with ε-moves that accept all reachable configurations of $\mathcal{P}$. In Algorithm 5, $\mathcal{A}_{post*}$ is obtained from $\mathcal{A}$ in two phases:

1. For each (p', γ') satisfying that $\mathcal{P}$ contains at least one rule of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma \rangle$, add a new state $q_{p',\gamma'}$.
2. Add new transitions to A according to the following saturation rules:

- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle$ and $p \xrightarrow{\gamma}{}^{*} q$ in the current automaton, add a transition (p', ε, q).
- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and $p \xrightarrow{\gamma}{}^{*} q$ in the current automaton, add a transition (p', γ', q).
- If $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and $p \xrightarrow{\gamma}{}^{*} q$ in the current automaton, first add $(p', \gamma', q_{p',\gamma'})$ and then $(q_{p',\gamma'}, \gamma'', q)$.

C A Proof of the Lemma 1

Proof. Since $\langle \mathcal{A}_i, p \rangle \mapsto \langle \mathcal{A}_j, q \rangle$, by **Definition** 9, there exists two configurations c and c', such that $c_0 = \langle \mathcal{A}_0, q_0(\mathcal{A}_0), \nu_{\mathbf{0}} \rangle \longrightarrow^{*} c = \langle \mathcal{A}_i, p, \nu \rangle \cdots \longrightarrow^{*} c' = \langle \mathcal{A}_j, q, \nu' \rangle \cdots$, where ν, ν' are clock valuations over $X(\mathcal{A}_i)$ and $X(\mathcal{A}_j)$ respectively.

1. Obviously, by the definition of reachability problem of NeTAs, $\langle \mathcal{A}_i, p \rangle \mapsto \langle \mathcal{A}_j, q \rangle$ implies both of TAs, $\mathcal{A}_i$ and $\mathcal{A}_j$, must appear at the top of stack. Since $\mathcal{A}_{post*}$ can recognize all reachable configurations from initial configuration $c_0 = \langle \bullet, \mathcal{A}_0 \rangle$, there must be 2 transitions $s_0 \xrightarrow{\mathcal{A}_i} s_1$ and $s_0 \xrightarrow{\mathcal{A}_i} s_1$, where $s_1, s_2 \in Q(\mathcal{A}_{post*})$.
2. By contradiction. Assume property 2 don't hold. With out loss of generosity, assume $(\mathcal{A}_i, q_0(\mathcal{A}_i)) \nrightarrow^{*} (\mathcal{A}_i, p)$. This implies the control location p of TA $\mathcal{A}_i$ can never be reached . Furthermore, $\langle \mathcal{A}_i, p \rangle$ can never appear at the top of stack, which contradicts the fact $c_0 = \langle \mathcal{A}_0, q_0(\mathcal{A}_0), \nu_{\mathbf{0}} \rangle \longrightarrow^{*} c = \langle \mathcal{A}_i, p, \nu \rangle \cdots$. Hence, property 2 must hold.
3. If $i = j = k$, then it's trivial that $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k \rightsquigarrow \mathcal{A}_j$ due to the reflexive of $\leftrightsquigarrow$ and $\rightsquigarrow$. If not this case, consider expanding the reachable path $c = \langle \mathcal{A}_i, p, \nu \rangle \cdots \longrightarrow^{*} c' = \langle \mathcal{A}_j, q, \nu' \rangle \cdots$. Since we focus on the TAs' reachability, we ignore the internal location change and timing behaviours of TAs. The reachable path can be transformed to a transition sequence $\omega_0 = \langle \mathcal{A}_{k_0}(i.e.\mathcal{A}_i) \rangle \cdots \xrightarrow{\phi_0} \omega_1 = \langle \mathcal{A}_{k_1} \rangle \cdots \xrightarrow{\phi_1} \cdots \xrightarrow{\phi_{n-1}} \omega_n = \langle \mathcal{A}_{k_n}(i.e.\mathcal{A}_j) \rangle \cdots$, where ω_k represent stack word and $\phi_k \in \{push, pop, internal\}$, for $0 \leq k < n$. Noticed that some reachable paths have useless "segment". If there exist $0 \leq a < b \leq n$, such that $\omega_a = \langle \mathcal{A}_{k_a} \rangle \cdots \xrightarrow{\phi_a} \cdots \xrightarrow{\phi_{b-1}} \omega_{k_b} = \langle \mathcal{A}_{k_b} \rangle \cdots$ and $\omega_a = \omega_b$, the partial transition sequence is useless "segment". It can be replaced with ω_a, keeping its original reachability. Assume the new transition sequence replaced all useless "segment" is $\omega_0 = \langle \mathcal{A}_{k_0}(i.e.\mathcal{A}_i) \rangle \cdots \xrightarrow{\phi_0} \omega_1 = \langle \mathcal{A}_{k_1} \rangle \cdots \xrightarrow{\phi_1} \cdots \xrightarrow{\phi_{m-1}} \omega_m = \langle \mathcal{A}_{k_m}(i.e.\mathcal{A}_j) \rangle \cdots$, where for $0 \leq i, j \leq m$, $\omega_i \neq \omega_j$. Notice the fact that for the new transition sequence there exists some $0 \leq k \leq m - 1$ such that:
   - for $0 \leq t \leq k$, $\phi_t \in \{pop, internal\}$;
   - for $k + 1 \leq t \leq m - 1$, $\phi_t \in \{push, internal\}$.

   By the definition of $\rightsquigarrow$ and $\leftrightsquigarrow$, we have $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k \rightsquigarrow \mathcal{A}_j$.
4. if $\mathcal{A}_i \rightsquigarrow \mathcal{A}_j$ don't hold, there must be some k such that $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k$, and $k \neq i$. This implies the fact that when the top symbol of stack is $\mathcal{A}_i$, it must be either popped out through "pop" transition or replaced with another TA

through “internal” transition. In either way, $\mathcal{A}_i$ muse reach one of its final locations;otherwise, it cannot be popped or replaced by the semantics of TAs. Therefore, $\exists q_f \in F(\mathcal{A}_i)$ s.t. $(\mathcal{A}_i, p) \rightarrow^* (\mathcal{A}_i, q_f)$.

D A Proof of the Lemma 2

Proof. The main idea is to construct a reachable path of $\langle \mathcal{A}_i, p\rangle \mapsto \langle \mathcal{A}_j, q\rangle$ from the above 4 properties. Due to the first property, $(s_0, \mathcal{A}_i, s_1) \in\rightarrow (\mathcal{A}_{post*})$ implies $\mathcal{A}_i$ can appear at the top of stack. We can construct a reachable path of $c_0 = \langle \mathcal{A}_0, q_0(\mathcal{A}_0), \nu_{\mathbf{0}}\rangle \rightarrow^* c_1 = \langle \mathcal{A}_i, q_0(\mathcal{A}_i), \nu_{\mathbf{0}}\rangle \cdots$ through all kinds of operations but ***Pop*** operation. Besides, configuration c_1 in the form $c_1 = \langle \mathcal{A}_{k_0}, p_{k_0}, \nu_{k_0}\rangle\langle \mathcal{A}_{k_1}, p_{k_1}, \nu_{k_1}\rangle \cdots \langle \mathcal{A}_{k_n}, p_{k_n}, \nu_{k_n}\rangle$,where for $0 \leq i \leq n, \mathcal{A}_{k_i} \in T(\mathcal{N}), p_{k_i} \in Q(\mathcal{A}_{k_i})$ and ν_{k_i} is a clock valuation over $X(\mathcal{A}_{k_i})$, should satisfy the requirement that for $0 \leq i \leq n$, if $\exists p_f \in F(\mathcal{A}_{k_i})$ *s.t.* $(\mathcal{A}_{k_i}, q_0(\mathcal{A}_{k_i})) \rightarrow^* (\mathcal{A}_{k_i}, p_f)$, then $p_{k_i} = p_f$. Note we can always do this, because when transferring to c_1 each time one TA is pushed into the stack, we can always check if topmost symbol in stack can reach one of its final locations. If so, wait until it reach one of its final locations. With second property, $(\mathcal{A}_i, q_0(\mathcal{A}_i)) \rightarrow^* (\mathcal{A}_i, p)$ implies $c_1 = \langle \mathcal{A}_i, q_0(\mathcal{A}_i), \nu_{\mathbf{0}}\rangle \cdots \rightarrow^* c_2 = \langle \mathcal{A}_i, p, \nu\rangle \cdots$, where ν is a clock valuation over $X(\mathcal{A}_i)$. Note the transitions from c_1 to c_2 involve only progress transition and intra-action of TA $\mathcal{A}_i$. Next we consider two different cases.

1. if $\mathcal{A}_i \rightsquigarrow \mathcal{A}_j$ holds, when the topmost symbol in stack is $\mathcal{A}_i$, we can always make the topmost symbol be $\mathcal{A}_j$ through all kinds of operations except *Pop* operation. Hence, $c_2 = \langle \mathcal{A}_i, p, \nu\rangle \cdots \rightarrow^* c_3 = \langle \mathcal{A}_j, q_0(\mathcal{A}_j), \nu_{\mathbf{0}}\rangle \cdots$. Note in the process of transferring to c_3, the symbols below topmost symbol $\mathcal{A}_i$ are still there and their locations are not changed.
2. if $\mathcal{A}_i \rightsquigarrow \mathcal{A}_j$ doesn't hold, there exist a k, $k \neq i$ s.t. $\mathcal{A}_i \leftrightsquigarrow \mathcal{A}_k \rightsquigarrow \mathcal{A}_j$, where $\mathcal{A}_k \in T(\mathcal{N})$. By the definition of $\leftrightsquigarrow$ and the specific requirement for c_1, we have $c_2 = \langle \mathcal{A}_i, p, \nu\rangle \cdots \rightarrow^* c_4 = \langle \mathcal{A}_k, q_0(\mathcal{A}_k), \nu\rangle \cdots$ through only *pop* and *internal* operations. Note, due to the fourth property, $\mathcal{A}_i$ can reach one of its final locations and can be popped out or replaced. For $\mathcal{A}_k \rightsquigarrow \mathcal{A}_j$, similarly we have $c_4 = \langle \mathcal{A}_k, q_0(\mathcal{A}_k), \nu\rangle \cdots \rightarrow^* c_5 = \langle \mathcal{A}_j, q_0(\mathcal{A}_j), \nu\rangle \cdots$ through all kinds of operations except *Pop* operation.

In general, in both cases we have $c_0 \rightarrow^* c_2 = \langle \mathcal{A}_i, p, \nu\rangle \cdots \rightarrow^* c_6 = \langle \mathcal{A}_j, q_0(\mathcal{A}_j), \nu_{\mathbf{0}}\rangle \cdots$. By second property, we have $c_6 = \langle \mathcal{A}_j, q_0(\mathcal{A}_j), \nu_{\mathbf{0}}\rangle \cdots \rightarrow^* \langle \mathcal{A}_j, q, \nu'\rangle \cdots$, where ν' is a clock valuation over $X(\mathcal{A}_j)$. Since $c_0 \rightarrow^* c_2 = \langle \mathcal{A}_i, p, \nu\rangle \cdots \rightarrow^* \langle \mathcal{A}_j, q, \nu'\rangle \cdots$, $\langle \mathcal{A}_i, p\rangle \mapsto \langle \mathcal{A}_j, q\rangle$ holds.

References

1. Li, G., Cai, X., Ogawa, M., Yuen, S.: Nested timed automata. In: Braberman, V., Fribourg, L. (eds.) FORMATS 2013. LNCS, vol. 8053, pp. 168–182. Springer, Heidelberg (2013)

2. Abdulla, P.A., Atig, M.F., Stenman, J.: Dense-Timed pushdown automata. In: Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS'12). IEEE Computer Society (2012), pp. 35–44 (2012)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput.Sci. **126**, 183–235 (1994)
4. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inf. Comput. **111**, 193–244 (1994)
5. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. J. Softw. Tools Technol. Transfer **1**, 134–152 (1997)
6. Bengtsson, J.E., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
7. Schwoon, S.: Model-checking pushdown system. Ph.D. thesis, Technical University of Munich (2000)
8. Bouyer, P., Dufourd, C., Fleury, E., Petit, A.: Updatable timed automata. Theor. Comput. Sci. **321**, 291–345 (2004)
9. Bouyer, P.: Forward analysis of updatable timed automata. Formal Methods Syst. Design **24**, 281–320 (2004)
10. Ausmees, K.: Zone-based reachability analysis of dense-timed pushdown automata. Student thesis, IT 12 034, Department of Information Technology, Uppsala University (2012)
11. Trivedi, A., Wojtczak, D.: Recursive timed automata. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 306–324. Springer, Heidelberg (2010)

Adopting Variable Dependency in Animation for Presenting the Behaviour of Process

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. Scenario-based formal specification animation can dynamically present the specification without translating it into executable program. The behaviours of the system defined in the specification are organized as sequences of processes. The user can observe a specific system behaviour by watching the "execution" of the sequence of processes. To present the "execution", each process is sequentially connected via data, which are generated based on the pre- and post-condition of the specification. However, the animation method does not provide the user with a chance of observing what happens inside a process. In this paper, we use a sequence of atomic predicate expressions to present the behaviour defined in a specific process. The atomic predicate expressions are organized based on the dependency of variables that are involved in the process. We define the variable dependency and illustrate the derivation of the dependency graph from the specification. The procedure to reorganize the atomic predicates based on the dependency graph is demonstrated with an example.

Keywords: Animation · Variable dependency · Formal specification · Process

1 Introduction

Specification animation is a technique for dynamically demonstrating the behaviour of the potential system through specification "executions" [1]. The execution can be implemented using different techniques. Most of the existing animation approaches automatically transform the formal specification into a program in an executable programming language and then execute the program to provide chances for the user to understand and validate the specification [2,3,10]. However, this approach can only deal with a subset of the formal notation in which

This work has been conducted as a part of "Research Initiative on Advanced Software Engineering in 2012" supported by Software Reliability Enhancement Center (SEC), Information Technology Promotion Agency Japan (IPA). It is also partly supported by the SCAT research foundation.

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 81–93, 2015.
DOI: 10.1007/978-3-319-17404-4_6

the formal specification is written, because formal specifications are generally not executable [4]. In our previous work [13], we proposed a new animation approach called *scenario-based animation*, which can directly deal with the execution of formal specifications without the need of the transformation. The basic unit in the animation is operational behaviour, which is usually defined by a specific relation between the input and output of the system. Each operational behaviour is defined as a sequence of independent operations called **system functional scenario** in the formal specification. The independent operations are defined as *processes* in the context of SOFL. The test data (or animation data), which can be automatically generated based on logical expressions derived from the specification, are used to connect processes for simulating the actual execution of the system functional scenario.

A process in the SOFL formal specification describes an independent relation between the input and output under a certain condition and usually presented as a predicate expression called **operation functional scenario**. It precisely defined the condition that should be satisfied by the input and output. A process can be realized as the basic functional unit in the SOFL specification. Different processes are integrated together to describe system behaviours. Although the scenario-based animation clearly presents how the processes are organized, it does not provide the user with a chance of observing what happens inside a process. In this paper, we use a sequence of atomic predicates, which can be derived from the operation functional scenario, to present the behaviour defined in a specific process. The sequence of the atomic predicates are decided based on the dependency of variables involved in the process.

In an executable program, the user can understand the function of the program by reading the statements one by one. This is because the statements in the program are organized in an execution order. The latter statements can not be executed until the previous statements have been executed. The value of variables in the program are changed by executing related statements. Therefore, the execution order indicates the dependency relations among the variables. The atomic predicates in process can not be executed. They only describe conditions that should be satisfied by the input and output data. There is no order between predicates to indicate which one should be presented first to user so that the user can understand what happens inside the process by reading the predicates sequentially. In order to reorganize the atomic predicates into a meaningful order, the dependency of variables involved in the process is adopted. We introduce how to build the dependency graph and how to reorganize the predicates based on the dependency graph.

The remainder of this paper is organized as fellows. Section 2 describes the scenario-based animation method as a background. Section 3 introduces the variable dependency in the context of formal specification and compares it with the variable dependency in the program. Section 4 demonstrates how the atomic predicates in a process is reorganized based on the dependency graph. Section 5 gives a brief overview of related work. Finally, in Sect. 6 we conclude the paper and point out future research directions.

2 Scenario-Based Animation

The scenario-based animation method is first proposed in our previous work [13]. In this section, we briefly introduce the idea of the animation method can give some definitions used in the rest of the paper.

The animation process includes the following three steps:

1. Deriving all possible *system scenarios* from the formal specification.
2. Extracting specific *operation scenario* for each process involved in a selected system scenario.
3. Generating data based on operation scenarios and "executing" system scenario using the generated data.

The basic unit in the scenario-based animation method is the system functional scenario. As mentioned previously, each system scenario presents a specific behaviour of the system. In order to observe all the potential behaviours of the system, we suggest that every possible system functional scenario defined in the specification should be animated. A system scenario defines a specific kind of operational behaviour of the system through a sequential executions of operations. It is usually presented to end users as a pair of input and output, that is, given an input, the result of a behaviour of the system results in a certain output. The definition of a system functional scenario is detailed in Definition 1.

Definition 1. *A* ***system functional scenario****, or* ***system scenario*** *for short, of a specification is a sequence of operations $d_i[OP_1, OP_2, \ldots, OP_n]d_o$, where d_i is the set of input variables of the behaviour, d_o is the set of output variables, and each $OP_i (i \in \{1, 2, \ldots, n\})$ is an operation.*

The system scenario $d_i[OP_1, OP_2, \ldots, OP_n]d_o$ defines a behaviour that transforms input data item d_i into the output data item d_o through a sequential execution of operations $OP_1, OP_2, \ldots, OP_n$. Actually, other data items are used or produced within the process of executing the entire system scenario but not being presented. For example, the first operation OP_1 in the system scenario receives the input data item d_i and produces a data item, which is the input data item of operation OP_2. Operation OP_2 cannot be executed without the output data item of OP_1. We call these data items *implicit data items.* In order to show the behaviour of system step by step in an animation, the implicit data items in system scenario should be presented explicitly. When presenting implicit data items explicitly is necessary, we use $[d_i, OP_1, d_1, OP_2, d_2, \ldots, d_{n-1}, OP_n, d_o]$ to present a system scenario, where d_1 indicates the output data item of OP_1 or input data item of OP_2.

To animate a specific system scenario, data are used to connect each operation involved in the scenario. Since the data is restricted by the pre- and post-conditions of process, the data present a real environment of the behaviour. The user and experts can observe the behaviour by monitoring the data.

When collecting input and output data for a single process, the operation functional scenarios of the process have to be extracted first. By operation functional scenario, we mean an predicate expression derived from the pre- and

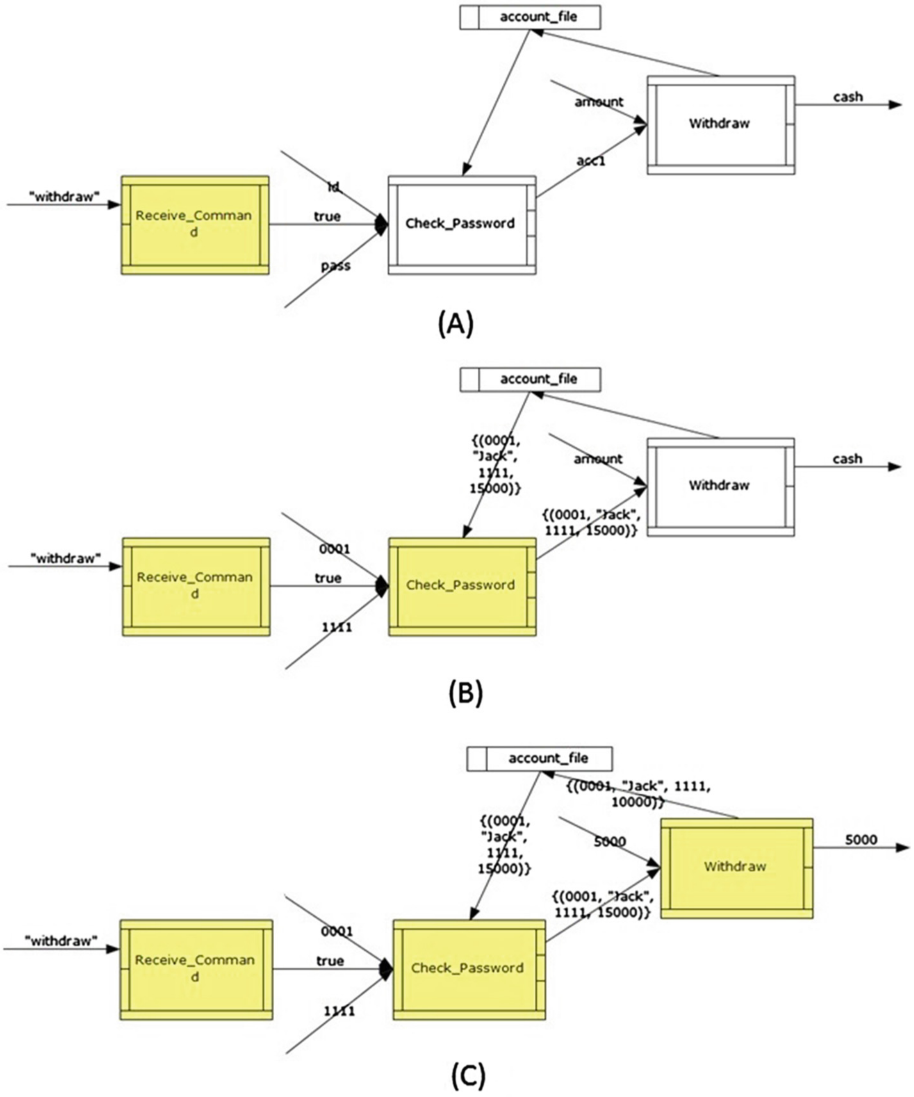


Fig. 1. Scenario-based animation of a simple ATM specification

post-condition of a process, which precisely defines the relation of a set of input and output data. Liu first gives a formal definition of operation functional scenario [12] and we repeat it here to help the reader understand the rest of this paper.

Definition 2. *Let $OP(OP_{iv}, OP_{ov})[OP_{pre}, OP_{post}]$ denote the formal specification of an operation OP, where OP_{iv} is the set of all input variables whose values are not changed by the operation, OP_{ov} is the set of all output variables* *whose values are produced or updated by the operation, and OP_{pre} and OP_{post} are the pre and post-condition of operation OP, respectively.*

Definition 3. *Let $OP_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \ldots \vee (C_n \wedge D_n)$, where each $C_i (i \in \{1, \ldots, n\})$ is a predicate called a "**guard condition**" that contains no output variable in OP_{ov} and $\forall_{i,j \in \{1,\ldots,n\}} \cdot i \neq j \Rightarrow C_i \wedge C_j = false$; D_i a "**defining condition**" that contains at least one output variable in OP_{ov} but no guard condition. Then, a formal specification of an operation can be expressed as a disjunction $(\sim OP_{pre} \wedge C_1 \wedge D_1) \vee (\sim OP_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (\sim OP_{pre} \wedge C_n \wedge D_n)$. A conjunction $\sim OP_{pre} \wedge C_i \wedge D_i$ is called an **operation functional scenario**, or **operation scenario** for short.*

The data used to connect processes can be generated by using the algorithm introduced in [13]. They are generated based on the operation functional scenario and satisfy the pre- and post-condition of related processes. Therefore, the generated data can present the intermediate status of the system scenario. The user can observe the corresponding system behaviour through monitoring the data. Figure 1 shows the animation of the formal specification defining a simple ATM system.

3 Variable Dependency

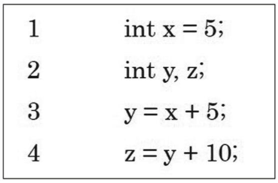
As shown in Fig. 1, the original scenario-based animation method can clearly presents the relation between different processes, and the data satisfying the pre- and post-condition provide the user a real environment in which the specification is running. However, the animation method does not reveal how the input of a process produces the output.

If a process is implemented into program, user can observe how the input produces the output by watching the execution of the program. In the execution, each statement in the program is executed sequentially. Unlike the program, the specification can not be executed, and there is no sequence between different predicate expressions included in the operation functional scenario. In order to present the behaviour inside the process, the predicates in operation scenario are reorganized to simulate the sequence of statements in the program. The reorganization is based on the dependency of the variables involved in the scenario.

3.1 Comparing with the Variable Dependency in Program

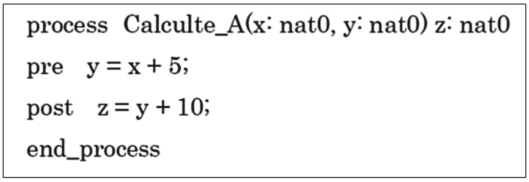
Variable dependency is widely used in program analysis [5–7]. A graph called *Dependence Graph* is generated to help the analysts understand the relation between different variables used in the program and check the data consistency. Some researchers use *program dependence graph* to do the analysis and optimization [8,9].

There are two differences between the variable dependency in program and the variable dependency in specification. The first difference is that the variable at the left side of a assignment is dependent on the variables at the right side



```
1        int x = 5;
2        int y, z;
3        y = x + 5;
4        z = y + 10;
```

Fig. 2. A simple program segment



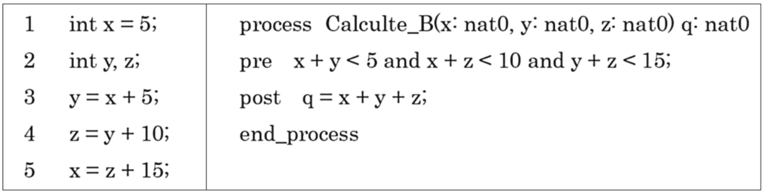
```
process  Calculte_A(x: nat0, y: nat0) z: nat0
pre   y = x + 5;
post   z = y + 10;
end_process
```

Fig. 3. A simple process specification

of a assignment in the program. Consider the program segment shown in Fig. 2. Three variables and two statements are included in the program segment, and the dependency relation between each variable is that "y" is dependent on "x", and "z" is dependent on "y". In the third line of the program, the value of variable "y" is determined by the value of "x". The value of "z" is determined by the value of "y" in the forth line. Note that both "y" and "z" are at the left side of assignment.

However, the variable dependency relation in specification is different from the case in the program. Use the process specification in Fig. 3 as an example, which contains three variables and the pre- and post-condition appears the same as the two statements in the previous program. The pre-condition indicates that "x" and "y" are dependent on each other rather than that "y" is dependent on "x". This is because the pre-condition is a predicate and only describes the condition that "x" and "y" should satisfy. There is no difference between "$y = x + 5$" and "$x = y - 5$" in term of the pre-condition. If the value of "x" is generated first in the data generation process, the value of "y" is based on the value of "x" and vice versa. Therefore, we can state that before the value of any variable is generated, the variables in the same predicate are dependent on each other.

The second difference between the variable dependency in specification and the variable dependency in program is that the variables in specification may be involved in *loop dependence*. Consider the program and specification shown in Fig. 4 as example. The program at the left side contains three variables "x", "y", and "z". According to the statements, "y" depends on "x", "z" depends on "y", and "x" depends on "z". It seems that the three variables are involved in loop dependence, however, the variable "x" in the third line and the variable "x" in the fifth line actually represent two different values.



```
int x = 5;
int y, z;
y = x + 5;
z = y + 10;
x = z + 15;
```

```
process  Calculte_B(x: nat0, y: nat0, z: nat0) q: nat0
pre   x + y < 5 and x + z < 10 and y + z < 15;
post   q = x + y + z;
end_process
```

Fig. 4. Example for loop dependence

In the other hand, the pre-condition of the process shown in the right side of Fig. 4 indicates that variable "x", "y", and "z" are dependent on each other. As long as the value of any variable is determined, the other two variables will depend on both this variable and each other, and the variable with determined value depends on non of other variables.

The two differences between the variable dependency in specification and program are caused by two reasons: first, the variables at the right side of assignment in the program are actually represent values, however, all the variables in specification represent undetermined values; second, the statements in program are executed one by one in a predetermined order and they change the value of variables, on the contrary, the predicates in the specification only describe the conditions that should be satisfied by the variables and do not have orders.

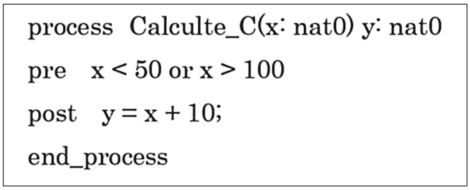
3.2 Variable Dependency in Specification

Since the variable dependency in specification is different from the variable dependency in program, the existing algorithms [5,6] for creating variable dependency in program can not be used here. To derive the variable dependency in specification, the following two principle must be satisfied:

- The variables in the same predicate must depend on each other.
- At least one output variable depends on input variables.

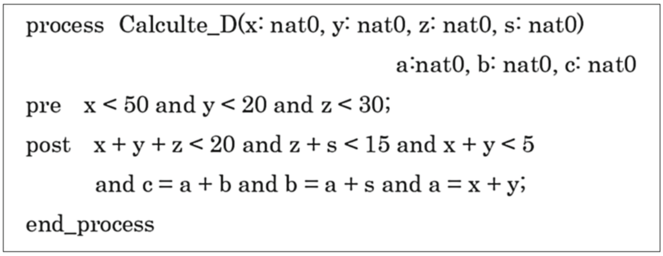
The derivation process can be roughly separated into two steps: decomposing operation functional scenario and building dependency graph. Decomposing the operation functional scenario is to translate the scenario $\sim OP_{pre} \wedge C_i \wedge D_i$ into an equivalent disjunctive normal form (DNF) with form $P_1 \vee P_2 \vee \ldots \vee P_n$. A P_i is a conjunction of atomic predicate expressions, say $Q_i^1 \wedge Q_i^2 \wedge \ldots \wedge Q_i^m$.

Each conjunction P_i in the DNF defines one possible situation that the operation functional scenario can be satisfied. For example, the operation functional scenario of the process shown in Fig. 5 is "$(x < 50 \vee x > 100) \wedge y = x + 10$". In the scenario, the disjunction "$x < 50 \vee x > 100$" is the pre-condition, "$y = x + 10$" is the defining condition, and there is no guard condition. The translated DNF of the scenario is "$(x < 50 \wedge y = x + 10) \vee (x > 100 \wedge y = x + 10)$". There are two conjunctions in the DNF. Each of them describes a possible combination of variables that satisfies the scenario. Since each conjunction in DNF is exclusive, the generated data in the animation can make only one conjunction



```
process  Calculte_C(x: nat0) y: nat0
pre   x < 50 or x > 100
post   y = x + 10;
end_process
```

Fig. 5. Example for scenario decomposition



```
process  Calculte_D(x: nat0, y: nat0, z: nat0, s: nat0)
                                    a:nat0, b: nat0, c: nat0
pre   x < 50 and y < 20 and z < 30;
post   x + y + z < 20 and z + s < 15 and x + y < 5
       and c = a + b and b = a + s and a = x + y;
end_process
```

Fig. 6. Example specification for illustrating dependency graph

to be true. In order to present the conjunction in a meaningful manner to the user, the atomic predicate expressions are reorganized based on the dependency of variables involved in the conjunction.

For each atomic predicate Q_i^m, each variable involved in it corresponds to a vertex in the dependency graph and there is an edge connecting each vertex. For example, the operation functional scenario of the specification in Fig. 6 can be translated to a DNF with nine atomic predicates: "$x < 50 \wedge y < 20 \wedge z < 30 \wedge x + y < 5 \wedge x + y + z < 20 \wedge z + s < 15 \wedge a = x + y \wedge b = a + s \wedge c = a + b$". Each variable in the predicates correspond to a vertex in the dependency graph and there is an edge connecting the variables involved in the same predicate. Figure 7 shows the dependency graph of the DNF. The upper side is the dependency of input variables and the lower side is the output variables.

4 Reorganizing Atomic Predicates in Operation Functional Scenario

The purpose of reorganizing atomic predicates in operation functional scenario is to arrange the predicates in a meaningful order so that they can present how the input of a process produce the output. The program can present the procedure clearly since the statements in the program are executed one by one and the value of variables are changed and decided after the execution of each statement. However, the atomic predicates in operation functional scenario do not have order and the value of variables are only required to satisfy the predicate. The reorganization of the atomic predicates tries to make the value of the variables

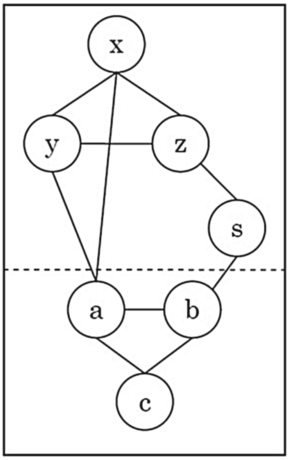


Fig. 7. Dependency graph of process "Calculate_D"

in latter predicates can be decided by the value of variables decided in previous predicates, so that the user can observe the predicates in an "*executionorder*". By execution order, we mean the relation between the previous predicate and the next one seems that they are executed one by one.

To reorganize the atomic predicates in an execution order, the variable dependency is adopted. Since the variable dependency graph of program is a tree structure, the value of lower level variables in the tree can be decided by higher level variables. The tree structure indicates the procedure that how the input produces the output. On the contrary, the variable dependency graph of specification is a graph with loop rather than a tree. This is because the variables in the same predicate depend on each other, and more than two variables in a predicate will make loops in the dependency graph. To make the predicates can be organized in an execution order, the variable dependency graph with loops should be reformed to a tree structure.

Stage 1. *Eliminate the vertex in the variable dependency graph containing only input variables of the process. The elimination makes the reminder of the graph be a connected graph and contain minimum number of loops. If the reminder graph contains loops, eliminate another vertex to make the reminder of the graph be a connected graph and contain minimum number of loops. The elimination terminates if the reminder graph is a tree. The the vertices are called "pre-decided variables"*

Note that in Step 1 the dependency graph includes only input variables of the process, this is because the value of input variables is generated first in the animation and they decide the value of output variables. To reform the dependency graph into a tree structure, the loops in the dependency graph have to be eliminated. Based on the algorithm in graph theory, the edges that struct loops are

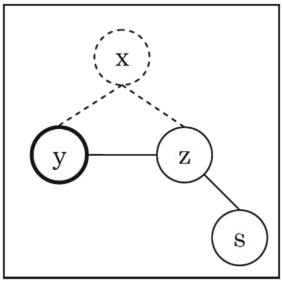


Fig. 8. Tree structure in dependency graph

usually deleted to eliminate the loops. Since only deleting edges can not indicate which variable should be the root of the true structure, this method is not appropriate for our reorganization purpose.

For example, the dependency graph in Fig. 7 contains a loop among variable "x", "y", and "z". If any edge in this loop is deleted, the graph will become a tree and any of the four variable can be the root of the tree. Based on the analysis of the process specification in Fig. 6, "x" or "y" should be the root of the tree so that the tree can be organized in an execution order. However, if the vertex "x" is eliminated and marked as pre-decided variable, the root of the tree can be decided based on the following step.

Stage 2. *Calculate the number of variables excluding pre-decided variables of each atomic predicates containing only input variables, where the number called the* ***freedom degree*** *of predicate. Select the variable in the predicate that contains pre-decided variables and has minimum freedom degree as the root of the tree structure in the dependency graph. Mark the selected variable as pre-decided variable.*

If the vertex "x" is eliminated in the dependency graph and marked as pre-decided variable, the predicate containing "x" and has minimum freedom degree is "$x + y < 5$". Based on Step 2, variable "y" should be the root of the tree. Figure 8 shows the tree structure after vertex "x" is eliminated. The dotted edges connecting "x" indicate that it is eliminated and the bold border of vertex "y" indicates that "y" is the root of the tree.

After reorganizing the input variables, the order of output variable should be decided. Unlike the dependency graph of input variables, there is no need to eliminate a vertex to break the loop in the dependency graph of output variables.

Stage 3. *Calculate the freedom degree of each atomic predicates. Select the variable in the predicate that contains pre-decided variables and has minimum freedom degree as the start point to traverse the vertices in dependency graph of output variables. Mark the selected variable as pre-decided variable.*

Stage 4. *Recalculate the freedom degree of each atomic predicates after an output variable is marked as pre-decided variable. Select the variable in the predicate* *that contains pre-decided variables and has minimum freedom degree as the start point to traverse the vertices in dependency graph of output variables. Mark the selected variable as pre-decided variable. Repeat the selection procedure until all the output variable has been traversed.*

Table 1. Reorganization of the atomic predicates

Step	Variable	Atomic predicate
1	x (pre-decided)	$x < 50$
2	y	$y < 20$
		$x + y < 5$
3	z	$z < 30$
		$x + y + z < 20$
4	s	$z + s < 15$
5	a	$a = x + y$
6	b	$b = a + s$
7	c	$c = a + b$

Table. 1 shows the steps and the final order of the atomic predicate in specification "Calculate_D" in Fig. 6. The steps in the table is following the stages described previously and the order of the atomic predicates presents the how the input proceeds to the output.

5 Related Work

Formal specification animation is an effective technique for the communication between users and developers, however, the scenario-based specification animation [13] can only present the system behaviour at process level. Liu and Wang introduced an animation tool called SOFL Animator for SOFL specification animation [10]. When performing animation, the tool will automatically translate the SOFL specification into Java program segments, and then use some test case to execute the program. In order to provide reviewers a graphic presentation of the animation, SOFL Animator uses Message Sequence Chart (MSC) to present the simulation of the operational behaviours. Since not all the specification can be translated to program, only part of the specification can be presented in this way.

Muhammand et al. [5] introduced an method for deriving variable dependency in the program. The method includes building a dependency graph. They used the dependency relations to check the program and found that checking program using variable dependency is more effective that checking all the variables in the program. In [8], Paul Anderson et al. introduced their tool called

CodeSurfer. This tool can automatically derive the program dependency graph and support the user to inspect the program based on the dependency graph. In this paper, the dependency graph is also used but for different purpose. It is used to clearly present the operation functional scenario of the process so that the use can understand what happens inside the process.

6 Conclusions and Future Work

In this paper, we describe how the use the variable dependency graph to present the atomic predicates of the process in a meaningful order. Since the original scenario-based animation method does not provide a chance for user to observe the behaviour of a specific process, presenting the predicates of a process in an execution order can let the user understand how the input variables proceeds to the output. We distinguish the difference between the variable dependency in formal specification and the variable dependency in program. We also demonstrate how to manipulate the dependency graph so that it can be used to reorganize the atomic predicates.

In the future, one of our major focus is to add a function to our existing tool to support the presentation of the sequence of the atomic predicates. It will be an additional component of the scenario-based animation tool. Our another focus is to find a way make the data generation algorithm in the original animation method more effective and efficient.

References

1. Miller, T., Strooper, P.: Animation can show only the presence of errors, never their absence. In: Proceedings of 2011 Australian Software Engineering Conference, pp. 76–88. IEEE CS Press (2001)
2. Waeselynck, H., Behnia, S.: B model animation for external verification. In: Proceedings of the Second IEEE International Conference on Formal Engineering Methods, pp. 36–45 (1998)
3. Hewitt, M.A., O'Halloran, C.M., Sennett, C.T.: Experiences with PiZA, an animator for Z. In: Bowen, J.P., Hinchey, M.G., Till, D. (eds.) ZUM '97: The Z Formal Specification Notation. LNCS. Springer, Heidelberg (1997)
4. Hayes, I., Jones, C.B.: Specifications are not (necessarily) executable. Softw. Eng. J. **4**(6), 330–339 (1989)
5. Sadi, M.S., Halder, L., Saha, S.: Variable dependency analysis of a computer program. In: Proceedings of 2013 International Conference on Electrical Information and Communication Technology, pp. 1–5 (2014)
6. Harman, M., Fox, C., Hierons, R., Hu, L., Danicic, S., Wegener, J.: VADA: a transformation-based system for variable dependence analysis. In: Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation, pp. 55–64 (2002)
7. Weiser, M.: Program slicing. IEEE Trans. Softw. Eng. **10**(4), 352–357 (1984)
8. Anderson, P., Reps, T., Teitelbaum, T.: Design and implementation of a fine-grained software inspection tool. IEEE Trans. Softw. Eng. **29**(8), 721–733 (2003)

9. Ferrante, J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987)
10. Liu, S., Wang, H.: An automated approach to specification animation for validation. J. Syst. Softw. **80**, 1271–1285 (2007)
11. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer (2004) ISBN 3-540-20602-7
12. 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)
13. Li, M., Liu, S.: Automated functional scenarios-based formal specification animation. In: Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC 2012), pp. 107–115. IEEE CS Press (2012)

Education and Verification

Using Alloy in Introductory Courses of Formal Methods

Shin Nakajima(✉)

National Institute of Informatics, Tokyo, Japan
nkjm@nii.ac.jp

Abstract. Compact and easy-to-learn educational material of core ideas in formal methods is prepared for students in software engineering courses. Although mathematical logic is usually employed to explore the basic ideas precisely and concisely, some students with limited background are not able to follow the contents. We adapt Alloy to *sugar wrap* logic, which makes it possible for students to learn the core ideas by experimenting with the tool. The proposed material covers model-oriented specification notations and SAT-based automatic formal verification methods. These are important subfields of formal methods in view of both theory and practice for software engineering courses.

1 Introduction

Industry needs software engineers familiar with formal methods as these are new technologies to raise reliability levels of software-intensive systems. Current software engineering courses, however, do not allocate enough time for teaching formal methods since the courses cover lots of topics in a limited amount of time. Compact and easy-to-learn educational material is needed.

The conventional sources of formal methods are divided into two cases; (a) strong indications of mathematical logic (cf. [6,13]), or (b) in-depth explanations of a particular method (cf. [1,3,16,28]). The first approach is quite orthodox in presenting the basic ideas of formal methods, and it requires adequate training in mathematical logic before understanding the contents. The second approach is good for courses whose focus is to provide technical details of one particular notation. "Learning by Doing" style material, reported successful [19], is also fallen in this category.

Academic software engineering (SE) courses, however, are to present the subject matter as a general principle, focusing neither on mathematical logic too much, nor on a specific notation. These SE courses provide students with scientific knowledge on software development. It is considered to follow the statement by Shaw [25]; software engineering is the branch of computer science, in which scientific knowledge is preferentially applied. Formal methods are such knowledge stocks on scientific methods of software development [10].

S. Nakajima—also affiliated with The Graduate University for Advanced Studies (SOKENDAI).

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 97–110, 2015.
DOI: 10.1007/978-3-319-17404-4_7

Although formal methods cover a wide variety of topics, they can be divided into several subfields; model-oriented specifications, property-oriented specifications, or, algorithmic verification methods such as logic model-checking to name a few. Model-oriented specifications, in particular, are adapted in various formal notations of both historical and practical importance, e.g. VDM [16], SOFL [20], Z [26], B-method [1], Event-B [3], and Alloy [15]. Their syntax, semantics and methodological aspects are divergent, but these notations share a set of common core concepts.

This paper introduces the educational material on model-oriented specification notations and automated formal verification methods. Instead of relying on mathematical logic in its bare form, the material employs Alloy [15] to encode, not all, but most concepts and allow quick feedback with the automatic analysis tool. The idea is similar to the material described in [23], which employs Maude for teaching property-oriented notations and term rewriting systems (TRS). The coverage of the topics is complementary to what this paper mentions.

This paper structured as follows; Sect. 2 presents a background to develop the proposed educational material, which is followed by a series of examples adapted from the material in Sects. 3–5. Section 6 reports our experience in using the material in courses at graduate schools. Section 7 concludes the paper with discussion on the future use of the material.

2 Context

As the demand is increased in industry, academic software engineering courses are expected to offer curricula including formal methods. Some successful courses focus on a specific formal notation, some of which adapt "Leaning by Doing" style of teaching [19]. Students in software engineering, however, are expected to have broad knowledge on the subject, formal methods. They, when work in software industry, may use formal notations different from the one taught at schools. Questions arise here what common core ideas are in formal methods and how these are presented to students.

Formal methods have a long history starting in 1970's, and a lot of specification notations have been proposed since then. They are sometimes categorized according to the basic concepts they share [27]; e.g. model-oriented or property-oriented. Such common concepts are what to be taught in a disciplined manner in academic software engineering courses. For example, some of property-oriented notations share the notion of algebraic specifications and term-rewriting [23].

The model-oriented approach is of practical importance, adapted in various formal notations such as VDM, SOFL, Z, B-method, and Event-B. They have successful applications to industrial software development. Their syntax, semantics and methodological aspects are divergent. These notations share a set of core concepts, the state-based specification style and notion of refinement.

In conventional courses focusing on a particular notation, students are less motivated to understand the common core ideas than the details with the syntax of the notation and features of its tool. Semantics are usually explained using mathematical logic, but understanding common ideas from them is not easy. Students are expected to be familiar with *various* mathematical concepts; LPF for VDM, Zermelo set theory for Z, or typed set theory for B-method and Event-B. These differences are important from the viewpoints of the mathematical foundation of the formal notations. However, software engineering students usually are more concerned with software designs than mathematical logic. There is a gap between the conventional ways of mathematical presentations and the common core ideas that software engineering students should know.

This paper introduces an alternative approach to teaching the common core ideas in model-oriented specification notations. The material adapts the notion of lightweight formal methods (LFM) [14], which makes the tool-assisted learning possible [18]. It can be said "teaching *heavyweight* formal methods (model-oriented specification notations) with a *lightweight* formal notation (Alloy)."

We adapted Alloy because of the three main reasons; (a) the logic behind Alloy, namely first-order relational logic with a built-in transitive closure operator, is adequate to represent most concepts in the subject matter, (b) Alloy adapts its syntax familiar to software engineers and thus mathematical logic is sugared-wrapped, and (c) descriptions in Alloy can be analyzed automatically, which provides quick feedback to the students.

In addition to the core concepts of model-oriented notations, the educational material includes topics relating to the automatic verification methods. Since the advent of the bounded model-checking (BMC) to use Boolean satisfiability (SAT) [8], there has been much progress on these methods. Therefore, understanding the basis of SAT-based formal verification is important for software engineering students who may have lots of occasions to use such related tools. We present verification problems in the Alloy notation. Alloy is considered a *high-level* language to encode SAT problems since the Alloy tool uses SAT solvers as its backend.

Note that the topics on the model-based notations and on the SAT-based verification are mostly independent. When the material is used in classes at school, the instructors may choose appropriate topics from the material by considering the amount of time allocated for the subject matter in their courses.

Unfortunately, the analysis with Alloy is *not* complete because of the bounded analysis to achieve full automation. There are certainly formulas not properly handled; such cases result in spurious counterexamples for the case of checking assertions or missing instances for the case of generating witnesses (Chap. 5 in [15]). We, however, decided to use Alloy and not interactive theorem-provers, although the latter fully covers first-order logic. Our conjecture is that automated analysis is preferable for software engineering students who are not familiar with proof tactics that requires comprehensive knowledge on mathematical logic. Student may further study mathematical logic for formal methods in detail once they get to know such core concepts. Therefore, the Alloy-based educational material is developed for the introductory courses.

3 A Brief Introduction of Alloy

The educational material starts with a brief introduction of Alloy [15]. This part uses a simple example of Composite pattern, which is one of the well-known design patterns of object-oriented programs [11]. Since software engineering students are supposed to be familiar with the design patterns, the example is suitable for introducing Alloy.

3.1 Specification of Composite Pattern

Figure 1 illustrates a class diagram to show the structural aspects of the Composite pattern. In Alloy snippets, the structural elements, such as Component, are defined as `sig` (signature). `Component`, an abstract class, is designated by `abstract sig`. It has a named field `parent` whose type is `Component`, and is accompanied with a multiplicity keyword `lone` to stand for "zero or one." `Component` has two concrete classes, `Composite` and `Leaf`. The `sig Composite` has a named field `children` to refer to a set of `Component` instances. The `links fact` asserts that `children` and `parent` are related. This `fact` must always be satisfied being applicable to all the instances.

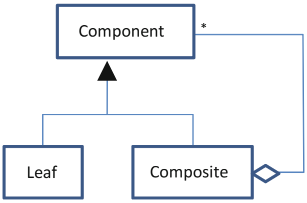


Fig. 1. Composite pattern

```
abstract sig Component { parent : lone Component }
sig Composite extends Component { children : set Component }
sig Leaf extends Component {}
fact links {
  all c : Composite, x : Component | x in c.children iff x.parent = c
}
```

The Composite pattern represents tree structures. There is a root `Composite`, namely `Root`, not to have any `parent`. `Root` is a named instance, and is defined in a manner similar to a singleton class in object-oriented programming style.

```
one sig Root extends Composite { no parent }
```

`Root` is a singleton (`one sig`) and inherits from `Composite` but has `no` parent.

3.2 Analysis with Scope-Bounded Search

Alloy provides two analysis commands, run and check. The run command allows us to enumerate configurations of instances to satisfy both the Alloy descriptions and the chosen predicate pred. We are now interested in obtaining a rooted tree, and thus define a predicate show to mention that all the instances can be traced from the sole Root. The operator *children denotes the reflexive-transitive closure of children relation.

```
pred show () { Component in Root.*children }
run show for <N>
```

The run command has an argument, <N> in the above example, to specify the scope of the analysis. Alloy searches for satisfiable configurations in the scope. It misses configurations outside the bound, and thus the search is not complete. For this example, Alloy tool returns appropriate witness instances in the specified scope of, for example, 3 or 5.

The check command of Alloy is employed to see whether a specified assertion is valid or not. The assertion acyclic below mentions that there is no Composite to be contained in a transitive closure of children from itself. In particular, we embed the predicate show above in the fact construct so that the check is conducted for rooted trees only.

```
fact { show[] }
assert acyclic { no c : Composite | c in c.^children }
check acyclic for <N>
```

The restriction due to the bounded search is also applicable to the check command, but this assertion can be shown satisfied in appropriate scopes.

4 Model-Oriented Specification Notations

This section introduces *core ideas* common to various model-oriented notations. It includes example Alloy snippets for illustrating the contents of the material.

4.1 State-Based Specification Style

Model-oriented notations employ state-based specification style for operations. The core ideas of (a) pre- and post-conditions and (b) invariants, are common to VDM, B-method, Event-B, and Z, but their roles in the specification descriptions are slightly different in each notation. The differences are concretely explained with Alloy snippets.

We use here BirthdayBook example [26] to show descriptions in each notation. All are presented in their original syntax, and their *intended* meanings are explained by translating them to Alloy. Note that the Alloy snippets, in some cases, are not exactly same as the original. For example, Alloy does not have partial mappings while VDM has. Furthermore, the Alloy snippets ignore the fact that VDM is based on three-valued logic or Kleene logic.

VDM Specification consists of type declarations and state definition [16], and VDM-like Alloy versions of `AddBirthday` are here. The first version is almost a direct translation from the VDM counterpart. The function `dom` is defined in Alloy utility module `util/relation`.

```
sig Name, Date {}      // token types in VDM
sig BirthdayBook {     // state in VDM
  known : set Name,
  birthday : Name -> Date  // no partial mapping in Alloy
}
pred invBB (b : BirthdayBook) { b.known = dom[b.birthday] }
```

The type of `birthday` is not appropriate as an Alloy description, because Alloy does not have partial mappings while VDM has. If we use this description for the analysis, some unexpected results will happen. The next version *simulates* the partiality in that the domain of `birthday` refers to a subset of `Name`. We introduced an auxiliary named field `names` to denote the domain of `birthday`.

```
sig Name, Date {}
sig BirthdayBook {
  known : set Name,
  names : set Name,          // simulating the partiality
  birthday : names -> Date  // using total mappings in Alloy
}
pred invBB (b : BirthdayBook) { b.known = dom[b.birthday] }
```

The VDM state definition is accompanied with an invariant `invBB`, which mentions that the domain of `birthday` is equal to `known`.

In VDM, an operation consists of pre- and post-conditions. Below, two Alloy predicates are defined. Both `birthday` and `known` are updated in the post-conditions (`postAddBB`). We follow a convention that the values in the post-state are referenced by an identifier with `’` such as `b’`. The `postAddBB` takes two arguments of `BirthdayBook`, and `b` and `b’` refer to its instance in the pre- and post-state respectively.

```
pred preAddBB (b : BirthdayBook, n : Name) { not(n in b.known) }
pred postAddBB (b,b’ : BirthdayBook, n : Name, d : Date)
   { b’.birthday = b.birthday ++ (n -> d) && b’.known = b.known + n }
```

Note that, in VDM, the specifier is responsible for the operation to satisfy the invariant. It, however, does not show explicitly the role of the invariant. Proof obligation, explained in Sect. 4.2, is the basis of the formal checking to see whether the operation specifications satisfy the invariant.

Invariant in Z, on the other hand, is a well-formedness condition that all the valid *models* should satisfy, and is a part of the `BirthdayBook` specification. Its role is explained in terms of the following Alloy snippets.

```
fact { all b : BirthdayBook | invBB[b] }
pred AddBB_Z (b,b' : BirthdayBook, n : Name, d : Date)
   { not(n in b.known) && b'.birthday = b.birthday ++ (n -> d) }
```

All the valid BirthdayBook instances satisfy invBB, which is specified with the fact above. The invariant in Z is regarded as an auxiliary condition added implicitly to the user-defined functional specification. The above Alloy snippets show that the named field known need not be updated because the invariant is satisfied for all BirthdayBook instances, and thus the changes in the birthday, described explicitly in AddBB_Z, are *propagated* to the known.

4.2 Proof Obligations

Proof obligation (PO) is a logic formula, defined in each specification notation, used to ensure the consistency of the descriptions. PO plays an important role in model-oriented notations since it provides correctness criteria of the declarative specifications. Although the state-based specification style is common, PO formula is different in each notation. We present some examples below.

The satisfiability PO of VDM [16] looks as follows if it is written in Alloy. Invariant in VDM is a required condition that the operation (post-conditions) should satisfy. The snippets show that the invariant is considered an assumption on the pre-state; invBB[b] comes in the antecedent of the formula.

```
assert VDMsatsfiability {
  all b : BirthdayBook, n : Name, d : Date | some b' : BirthdayBook |
     preAddBB[b,n] && invBB[b] => postAddBB[b,b',n,d] && invBB[b']
}
```

Unfortunately, in Alloy, the check command returns counterexamples while the assertion can be ensured valid by inspection. These are actually spurious counterexamples due to the bounded search of Alloy.

The proof obligation in B-method takes a different form. B-method adapts weakest precondition semantics [1], and requires predicate transformers to represent PO. When Alloy pred construct represents B-method predicate, the predicate transformer is higher-order and Alloy cannot represent it. Alternatively, we construct PO manually by following the B-method semantic rules.

```
assert Bfeasibility {
 all b : BirthdayBook, n : Name, d : Date |
  invBB[b] && preAddBB[b,n] => invBB[mkBB[b.known+n,b.birthday++(n->d)]]
}
fun mkBB (x : set Name, y : Name->Date) : BirthdayBook {
  { b : BirthdayBook | b.known = x && b.birthday = y }
}
```

The check command ensures that the assertion is valid. No spurious counterexample was generated.

The PO in VDM is close to what the human engineers imagine from the state-based specification styles; it ensures the existence of a post-state for all the valid pre-states. Contrarily, constructing PO in B-method requires a *pre-processing* or *symboic execution* based on the weakest pre-condition semantics. The resultant formula, using ∀-quantifier only, is simpler than the PO formula of VDM that has ∃-quantifier. Students may intuitively understand such differences by using Alloy to check those formulas.

4.3 Refinement

Refinement is a concept that software engineering students are not familiar with. Although it is sometimes explained in relation to an intuitive stepwise software development method, the notion of refinement is more than that. Furthermore, the correctness of refinement is ensured to discharge relevant proof obligations (PO). When using tools such as RODIN, for Event-B, these logical formulas are generated and proved automatically[1]. Although such an automation is inevitable in view of tool users, the educational material must explain the core ideas explicitly, in particular the rational of PO formulas in each formal notation compactly and precisely.

The educational material sees the refinement from two viewpoints. The first is concerned with the role of refinement in the correct-by-construction (CxC) software development. It discusses the methodological aspects of refinement, and includes the classical notion of the stepwise development method. The second view looks at the refinement from the simulation relations, a mathematical basis of rigorous semantics and proof obligations. Although there are several simulation relationships from mathematical viewpoints, each specification notation, other than Z or Alloy, has its own built-in refinement relationship.

Those students, in courses to teach a particular notation other than Z or Alloy, may lose a chance to know that there are several variations in the notion of refinement. Although such variations might be considered an advanced topic or too detailed, these are so important that the differences should be taught in the introductory courses on formal methods. In Z or Alloy, which does not provide any built-in refinement PO, we have to write down formulas to express PO for refinement checking by ourselves. Knowing the variations is inevitable to conduct refinement checking. As a demonstration, the educational material contains the Alloy snippets for checking the refinement of a simple BirthdayBook example (Sect. 1.5 in [26]).

The methodological views on the refinement are important before going into the technical details. Refinement in model-oriented notations historically follows the Hoare-style data refinement [12]. An initial specification is refined towards concrete, executable programming language constructs. It is called data reification in VDM [16] or vertical refinement in general. An alternative style of usage, the horizontal refinement or superposition refinement [4], is possible in Event-B [3]. We can construct descriptions by adding new features on the existing

[1] We switch to the interactive proof when the automatic prover fails.

ones, which allows an incremental style of building specifications. It is particularly useful to construct requirements in a CxC way.

The *superposition* refinement requires the notation, Event-B, to adapt guard or enabling conditions in place of preconditions as in VDM. In vertical refinement, the precondition of the concrete operation is weaker than the abstract one. Oppositely, the guard condition of the concrete event is stronger than the abstract one. These differences manifest themselves in proof obligations of each specification notation that has built-in refinement PO. Such technical differences particularly need to understand when we study B-method and Event-B, two notations in the same family.

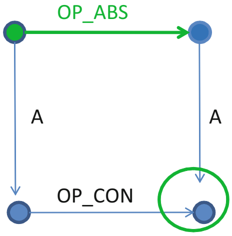


Fig. 2. Forward refinement

Mathematical notion of the refinement is defined in terms of simulation relationship between the abstract operation (or event) and concrete one. Although two simulation rules, forward and backward, are formulated [12], the notations such as VDM, B, and Event-B, adapt the forward simulation rule only (Fig. 2). The Alloy snippets, which follow Bolton's approach [7], help students understand the difference of two simulation relations. Basically, the Alloy snippets are concrete description of forward and backward simulation rules, which is sometimes informally explained in terms of diagrams such as Fig. 2.

Note that it should be pointed out here that two rules do not come from theoretical interests but are important in view of practice. Both rules are sound, but neither is complete [12]. Students must know in what situation the forward rule is not applicable although the refinement is intentionally correct. The educational material includes Alloy snippets of Phoenix-Apollo Theaters problem [28] to explain how the two simulation rules are applied to show the correctness of refinement. When checking if Apollo is a refinement of Phoenix, the forward rule is not applicable, but the backward rule is effective.

5 SAT-Based Formal Verification

The educational material contains (a) state-transition diagram and bounded model-checking (BMC), (b) software model-checking based on the BMC, and (c) automated test case generation for checking programs. These are independent of model-oriented specification notation, but are important topics in software engineering courses.

5.1 Bounded Model-Checking

Logic model-checking is now matured enough to be used in industry. Some textbooks (cf. [13]) also include the topic. Among various model checking algorithms, a bounded model-checking (BMC) method encodes transition relations as clauses in propositional logic and employs SAT solver to show the validity. BMC is sometimes superior to other model-checking algorithms in view of scalability to find counterexamples.

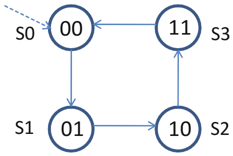


Fig. 3. Two bit counter

In the material, a simple historical example, 2-bit counter (Fig. 3) [8], illustrates the basic idea, which is followed by a bounded model-checking method of general Linear Temporal Logic (LTL) formulas. Here are the Alloy snippets for the 2-bit counter problem. `Bit` denotes values stored in each state (`State`). For example, the state `S1` has value of `01` in Fig. 3, which is represented in the snippets as $\texttt{S1.value} = \texttt{One}- > \texttt{Zero}$.

```
abstract sig Bit {}
one sig Zero, One extends Bit {}
abstract sig State { value : Bit -> Bit }
one sig S0, S1, S2, S3 extends State {}
fact {
     S0.value = Zero->Zero && S1.value = One->Zero
  && S2.value = Zero->One  && S3.value = One->One
}
```

The transition relations are kept in `Trans`. For example, the transition from `S0` to `S1` is represented as $\texttt{S0}- >\texttt{S1}$.

```
pred Init (s: State) { s = S0 }
one sig Trans { t : State -> State }
pred T (s,s' : State) { s->s' in Trans.t }
fact { Trans.t = S0->S1 + S1->S2 + S2->S3 + S3->S0 }
```

We here consider a safety property, $\Box\neg(\texttt{One}\!-\!\!>\!\texttt{One})$, which denotes that the value of the counter is always other than $\texttt{One}\!-\!\!>\!\texttt{One}$. BMC is a checking method to see if a formula of the form $System \wedge \neg Property$, a conjunction of *System* and a negation of *Property*, is valid. If satisfied, then the *Property* is violated and the obtained assignments constitute a counterexample. For the case of the 2-bit counter, because $\neg(\Box\neg(\texttt{One}\!-\!\!>\!\texttt{One})) = \Diamond(\texttt{One}\!-\!\!>\!\texttt{One})$, the check is to see if there is some state `s` to satisfy `P[s]`; the predicate `P[s]` is defined as below.

```
pred P (x : State) { x.value = One->One }
pred BMC (x1,x2,x3,s4 : State) {
     Init[x1] && T[x1,x2] && T[x2,x3] && T[x3,x4]   // System
  && (P[x1] || P[x2] || P[x3] || P[x4])    // negation of Property
}
run BMC for 4
```

The above Alloy command with the scope bound 4 returns a counterexample while it misses the violation for all the transition sequences with the length ≤3. The tool experiment helps students understand how the search bound is sensitive to the BMC results.

5.2 Other SAT Applications

As discussed above, bounded model-checking is itself an interesting subject and is a typical example of applying SAT technology to checking software artifacts [24]. The idea is adapted in software model-checkers, which works on source program codes of C or Java. Such an SBMC tool[2] encodes potential execution paths of a program in logic formula. Furthermore, in automatic test case generation, specification-based testing (SBT) method [9] employs pre- and post-conditions, or design by contract (DbC) of procedure.

BMC, SBMC, and SBT all make use of satisfiability checking of logic formula. The differences are what information is encoded; transition relations of an automaton for BMC, potential execution paths of a program for SMBC, and DbC of an operation for SBT. Students can experiment with the Alloy tool to learn that satisfiability checking is a common basis.

Last, a modern way of using model-checkers often relies on abstraction so as to make the state space to be of adequate size. It is called abstraction-aided verification (AAV) to make it clear the importance of *abstraction*. AAV is based on a method to obtain an abstract transition system from a given concrete system. Calculating abstract transition relation is basically a satisfiability checking and thus the problem is presented in the Alloy notation; a satisfied relation between two abstract states becomes a transition between these. In the educational material, we borrow the file transfer example from the Event-B Book (Chap. 4 in [3]). With AAV, we can show the correctness of the file transfer protocol for files with any length of `n`. Without abstraction, we only check the protocol for a specified concrete value for `n` such as `5` or `10`.

AAV is an established method, but the information can be found, today, in research papers only. The Alloy-based material may *de-mystify* the AAV method so that the students may understand there is no magic in them. The students also understand why AAV tools sometimes consume lots of computing power; the satisfiability checker is repetitively called to calculate abstract transition relations in order to construct an abstract transition system.

[2] hereafter called SBMC to stand for software bounded-model checker.

6 Course Experience

The educational material was publicly available in a form of a textbook [22] (about 200 pages), which is supposed to be used in a one-semester course. Course is offered at The Graduate University for Advanced Studies (SOKENDAI) every other year from 2009, and at Tokyo Institute of Technology (Tokyo Tech.) every year from 2012.

At SOKENDAI, the course students have their jobs in industry to be part-time working for their degrees in software engineering. Some of them had some prior experience in one or a few formal notations, A particular student working in industry had been involved in an industry-academia consortium to conduct feasibility studies of formal methods (e.g. Event-B) with “Learning by Doing” style activities. He commented that the educational material motivated engineers to learn general ideas of formal methods if they already had experience in at least one formal notation. Software engineers in industry are so busy that they do not have enough time to *identify* such common cores from a lot of existing material.

At Tokyo Tech., a software engineering course focusing on formal methods is offered as a part of the *enPiT* program[3] supported by MEXT[4]. It is designed for students who plan to find their careers in software industry. Most of the students is interested more in programming and using advanced software tools than in paper and pencil games (mathematical logic) or drawing diagrams (UML). They do not find any difficulty in using Alloy although the course allocates only two lectures, total of three hours, for teaching the basics of Alloy. We found a *side effect* of the material in that it turns out to be a good course of Learning by Doing for Alloy, which we did not intended. The material contains many Alloy examples of functional specifications and the proof obligations can be understood as a set of typical formulas for the correctness conditions. These can serve as a good reference for students when they use Alloy for describing and checking their own software design afterwards.

Last, it is not easy to evaluate the effectiveness of the educational material in an objective manner. Ideally, it is desirable to run two courses simultaneously and make comparisons, in which one course uses the Alloy-based proposed material and the other may follow conventional way of teaching, probably more on the mathematical logic. We unfortunately do not have such resources up to now.

7 Discussions and Conclusion

This paper described tool-assisted teaching of formal methods in software engineering courses. It provides an alternative way of teaching; instead of relying on mathematical logic in its bare form, we used Alloy to encode the core concepts and allowed quick feedback with the automatic analysis tool. Our conjecture is that automated analysis is preferable for software engineering students who are

[3] Education Network for Practical Information Technologies, http://www.enpit.jp.

[4] Ministry of Education, Cultural, Sports, Science and Technology, Japan.

not familiar with proof tactics that requires comprehensive knowledge on mathematical logic. Note that we do not say that mathematical logic is not needed anymore. Student may start studying mathematical logic in detail once they get to know such core concepts. In a sense, the Alloy-based educational material is expected to motivate them to study mathematical logic.

As pointed out in [15,18], tool-assisted learning helps students understand abstract concepts in software using analyzable concrete descriptions, thereby enhance their ability of abstractions. The educational material proposed in this paper shares the philosophy behind some textbooks in that tool-assisted learning was effective to show abstract concepts in a concise manner. There are, indeed, such textbooks, for example, in concurrent and distributed programming [5,21]. The topics are presented with concrete descriptions that are analyzable by model-checkers. This view is also shared with the material to use Maude for teaching property-oriented notations [23].

The proposed material focuses on learning the common core concepts, and does not provide further aspects relating to formal methods. Apparently, it is important to have some modeling experience with formal methods, which may be the focus of the Learning by Doing approach. These two approaches are, in our view, complementary. Therefore, we may suggest a combining course where the Learning by Doing method using a particular formal notation is augmented with the material discussed in this paper. We actually prepare such a course in which Event-B/RODIN is used for the Learning by Doing part. As mentioned in Sect. 4.3, RODIN tool encapsulates technical details of the refinement proof obligations. It is instructive to use the material discussed in this paper for students to learn the notion of refinement in reasonably details.

As we see, in Western countries, the formal methods are not just theoretical computer science, but have strong connections with practices (cf. [2,14,17]). On the other hand, in non-Western countries, formal methods are *imports* and often considered too theoretical. The motivation of the educational material can be said to *de-mystify* the formal methods. We hope that the material would help organizing similar courses in schools, not only in Japan, but also in non-Western countries. The contents are publicly available as [22], and sharing the teaching experience with us is really appreciated.

References

1. Abrial, J.R.: The B-Book - Assgining Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.R.: Formal methods in industry - achievements, problems, future. In: Proceedings of ICSE 2006, pp. 761–767 (2006)
3. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press, Cambridge (2010)
4. Back, R.-J., Sere, K.: Superposition refinement of reactive systems. Formal Aspects Comput. **8**(3), 324–346 (1995)
5. Ben-Ari, M.: Principles of Concurrent and Distributed Programming, 2nd edn. Addison-Wesley, Boston (2006)

6. Bjørner, D.: Software Engineering (three volumes). Springer, Berlin (2006)
7. Bolton, C.: Using the alloy analyzer to verify data refinement in Z. ENTCS **137**, 23–44 (2005)
8. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001)
9. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Proceedings of FME'93, pp. 268–284 (1993)
10. Dijkstra, E.W.: The humble programmer - ACM turing award lecture. Commun. ACM **15**(10), 859–866 (1972)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
12. He, J., Hoare, C.A.R., Sanders, J.W.: Date refinement refined - resume. In: Proceedings of ESOP'86, pp. 187–196 (1986)
13. Huth, M., Ryan, M.: Logic in Computer Science, 2nd edn. Cambridge University Press, Cambridge (2004)
14. Jackson, D., Wing, J.: Lightweight formal methods. In: Saidian, H. (ed.) An Invitation to Formal Methods, IEEE Computer (1996)
15. Jackson, D.: Software Abstractions - Logic, Language, and Analysis, revised edn. The MIT Press, London (2012)
16. Jones, C.: Systematic Software Development with VDM, 2nd edn. Prentice Hall, New York (1990)
17. Jones, C.: A rigorous approach to formal methods. In: Saidian, A. (ed.) An Invitation to Formal Methods, IEEE Computer (1996)
18. Kramer, J.: Is abstraction the key to computing? Comm. ACM **50**(4), 37–42 (2007)
19. Larsen, P.G., Fitzgerald, J.S., Riddle, S.: Learning by Doing: Practical Courses in Lightweight Formal Methods using VDM++, CS-TR-992. University Newcastle upon Tyne, Tyne (2006)
20. Liu, S.: Formal Engineering for Industrial Software Development using the SOFL Method. Springer, Edinburgh (2004)
21. Magee, J., Kramer, J.: Concurrency - State Models & Java Programming, 2nd edn. Wiley, Chichester (2006)
22. Nakajima, S.: Introduction to Formal Methods - Logic-Based Software Development Methods (in Japanese), Ohm-sya (2012)
23. Ölveczky, P.C.: Teaching formal methods based on rewriting logic and maude. In: Gibbons, J., Oliveira, J.N. (eds.) TFM 2009. LNCS, vol. 5846, pp. 20–38. Springer, Heidelberg (2009)
24. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in SAT-based formal verification. J. STTT **7**(2), 156–173 (2005)
25. Shaw, M.: Whither software engieering education?, an invited talk at IEEE CSEE&T 2011, Honolulu (2011)
26. Spivey, J.M.: The Z Notation - A Reference Manual. Prentice Hall, Englewood Cliffs (1992)
27. Wing, J.: A Specifier's Introduction to Formal Methods. IEEE Comput. **23**, 8–24 (1990)
28. Woodcock, J.C.P., Davies, J.: Using Z - Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)

Automatic Verification for Later-Correspondence of Security Protocols

Xiaofei Xie[1], Xiaohong Li[1](✉), Yang Liu[2], Li Li[3], Ruitao Feng[1], and Zhiyong Feng[1]

[1] School of Computer Science and Technology, Tianjin University, Tianjin, China
{xiexiaofei,xiaohongli,rtfeng,zyfeng}@tju.edu.cn

[2] School of Computer Engineering, Nanyang Technological University, Singapore, Singapore
yangliu@ntu.edu.sg

[3] School of Computing, National University of Singapore, Singapore, Singapore
li-li@comp.nus.edu.sg

Abstract. Ensuring correspondence is very important and useful in designing security protocols. Previously, many research works focus on the verification of former-correspondence which means "if the protocol executes some event, then it must have executed some other events before". However, in some security protocols, it is also important to ensure the engagement of some events after an event happens. In this work, we propose a new property called later-correspondence, which is very useful for e-commerce protocols. The applied π-calculus is extended to specify the protocols. A simplified intruder model is proposed for modeling the intruder capabilities which includes the malicious behaviors of both protocol agents and intruders. The later-correspondence is verified based on the Labeled Transition System (LTS) using model checking. In order to avoid the states explosion, we limit the number of protocol sessions and reduce most of the useless messages from the intruder knowledge with message pattern filtering. We implement our method in a model checker PAT [23] and the verification results show that our method can verify later-correspondence in an effective way.

1 Introduction

With the development of network technology, Internet is heavily involved in our life and work. Meanwhile, security protocols play a very important role in the Internet security. However, the design of protocols is error-prone and it is hard to find security flaws with the traditional testing methods [24]. Analyzing protocol properties through formal methods thus becomes an effective approach. In recent years, there has been more and more research works focusing on formal verification for security protocols, such as belief logic [1], theorem proving [6], strand space [4,5] and model checking [2,3]. Some automatic verification tools have been developed, for example Athena [7], ProVerif [8], Murphi [12], AVISPA [9]

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 111–126, 2015.
DOI: 10.1007/978-3-319-17404-4_8

and NRL [10] etc. These tools can verify security protocols efficiently regarding to secrecy and authentication properties.

As the basic property of security protocol, authentication has been studied extensively. One effective method is to use correspondence [14,18,19,22,24,25]. Woo and Lam firstly defined the correspondence assertion [25] to describe correspondence property. The general form of correspondence is like that: “if the protocol executes some event, then it must have executed some other events before” [24] which we called former-correspondence. The form of the later-correspondence is “if a protocol executed some events then it will execute some other events later”. Now there is a lot of research about the verification of former-correspondence while less research on the later-correspondence. Not only later-correspondence can be used to verify authentication of security protocols, but also can be used to verify that the messages of the protocol have been sent and received in the expected order [24].

Recently, It is important to verify later-correspondence. For example, in some electronic commerce protocols, it is necessary to guarantee the property “if the purchaser has made a payment, then the seller will send goods later”. This is a typical example of later-correspondence, but it can’t be verified through former-correspondence. The flaw is like that because of some malicious attacks, the event e_2 will not execute after executing the first event e_1. The kind of flaw is easy to be found through verification of later-correspondence and the designers can revise the protocol on time.

Comparison with Former-Correspondence. Suppose the valid run of one protocol is like $\dots e_1 \dots e_2 \dots$ A process P satisfies later-correspondence if and only if for every run of protocol, there is an event e_2 happens after event e_1 happens. As mentioned above, former-correspondence is the general correspondence: “if the protocol executes some event, then it must have executed some other events before”. The later-correspondence focus on whether there will be one event execute after the specific event rather than executing before the event. The later-correspondence can be verified through opposite of the former-correspondence in some situations, but it can’t be replaced in some other situations.

To verify the later-correspondence: after e_1 executes, there will be e_2 to execute. We will analysis whether it can be transformed into the former-correspondence: before e_2 executes, there has been e_1 executed. The traces of the protocol running have many cases and the result is listed in Table 1.

Table 1. Analysis of different cases

	Former-correspondence result	Later-correspondence result
$\dots e_1 \dots e_2 \dots$	True	True
$\dots e_1 \dots$	True	False
$\dots e_2 \dots$	False	True
$\dots$	True	True
$\dots e_2 \dots e_1 \dots$	False	False

The result shows that later-correspondence cannot be replaced by former-correspondence entirely. So the research about later-correspondence is significant and necessary. But there are some particular problems in verifying later-correspondence. For example, some exceptions that an attacker blocks all communications or the hardware is broken after the occurrence of event e_1 could prevent e_2 from occurring. This dissatisfies the later-correspondence. In the paper, we pay more attention to find the logic flaws in the design of the protocols. So some constraints are raised to avoid these invalid attacks. In the paper, we assume the hardware and network are always healthy and ignore the exceptions. In the attacker model, we restrict that the attackers cannot just block the messages. But they can replace the correct messages, which means the attacker can block the correct message and send another one.

This paper gives a formal definition of later-correspondence and proposes a method for verifying later-correspondence based on model checking technology. The rest of the paper is organized as follows. In Sect. 2, we discuss the related work. We present our main works which include syntax, operational semantics and verification in Sect. 3. In Sect. 4, we analysis the experimental results. Finally we draw a conclusion and give a brief introduction about future work in Sect. 5.

2 Related Works

The correspondence and authentication have been extensively studied. Using CSP and automatic tool FDR, Lowe modeled and analyzed authentication of Needham-Schroeder public key protocol [13] and found a flaw [2]. In paper [3], we also used CSP and FDR to model BGP protocol and verify the correspondence. However, CSP has a limited description capacity. When using CSP/FDR to verify protocols, a manual intruder model is needed. The method in this paper is based on the applied π-calculus [27] which can support more cryptographic primitives (including encryption,decryption and hash functions). Besides this, we also put forward an intruder model which can be automatically constructed.

Gordon and Jeffrey developed Cryptyc which can be used to describe fresh values and keys. Based on type system, Cryptyc system also can verify authentication of protocols [14–16]. In [17], based on Gordons theory Bugliesi drawn on a translation of well-typed -spi protocols into valid Cryptyc protocols. Then Bugliesi developed a type system for authentication protocols which need to be built upon a tagging scheme [18]. Although Cryptyc is mainly applied in shared-key protocols and public-key protocols. It mainly supports the basic encryption and decryption cryptographic primitives. With constructors and destructors, our method can model more cryptographic primitives, such as hash, signature etc.

In paper [19], Cremers proposed a general trace model to verify various formal definitions of authentications. Since this model is abstract, it does not suffice to prove protocols (in) correct. Due to the limitation of specifying security properties with LTL, Corin extended the LTL as PS-LTL for specifying security properties, and present a decision procedure to check a fragment of PS-LTL against symbolic traces for limited session protocols. In addition, they demonstrated the procedure was sound and complete [20]. The later-correspondence can be specified though *always* and *eventually* in LTL seemingly, but LTL has some limitations. The LTL is built up from propositional variables generally, but the later-correspondence is based on *event* which may contain arguments. This arguments will be instantiated as actual values in different runs. Another limitation is LTL can not specify the relationship about the number of occurrences of event. For example, one property like that the number of occurrences of e_1 is greater than, or equal to,the number of occurrences of e_2. Similar to the former-correspondence in Proverif, we extend it and formalize the later-correspondence in the paper.

Based on Athena and Scyther, Schmidt designed a verification tool Tamarin which models security protocols as multiset rewrite systems and describes properties as first-order logic formulas. Besides, they designed a novel constraint-solving algorithm to verify properties for unlimited session protocols [21]. Luu has developed a protocol verification tool SEVE in PAT (Process Analysis Toolkit) model checker. The verifier can also automatically build intruder model. It can verify secrecy, authentication, anonymity and receipt freeness through model checking technology [22]. However, the intruder model constructed by SEVE is simple and has low capacity. We also do model checking in the trace model which is similar with SEVE. The difference is that we extended the applied π-calculus [27] and our intruder model is more effective.

Blanchet developed the protocol verifier ProVerif [8] and added event operator and destructor in applied π-calculus [27]. ProVerif can verify the correspondence for security protocols automatically [24]. Whereas, it can only verify former-correspondence, but cannot be used to verify later-correspondence. On the basic of ProVerif, we define the later-correspondence and verify it in the trace model. ProVerif has been proven very effective in terms of security protocol verification. Based on the input language of ProVerif and we implemented the automatic tool SPChecker in PAT. Our method can verify later-correspondence effectively.

3 Modeling and Verification for Security Protocol

Figure 1 shows the method for modeling the protocols and the verification of later-correspondence. The protocols are formalized with the extended π-calculus [27] of ProVerif [8] and they are processed by the compiler. The intruder knowledge generator will construct the messages of intruder automaticly. With the intruder knowledge the π-calculus process can be transformed into LTS model based on the operational semantics. Then the later-correspondence can be verified though the on-the-fly refinement checking algorithm. If it finds an attack, one counter-example will be generated. The subsections will introduce every part in Fig. 1.

3.1 Syntax

Figure 2 gives the syntax of the terms and processes, most of them are based on π-calculus [27]. We only introduce the part which is extended by ProVerif [24].

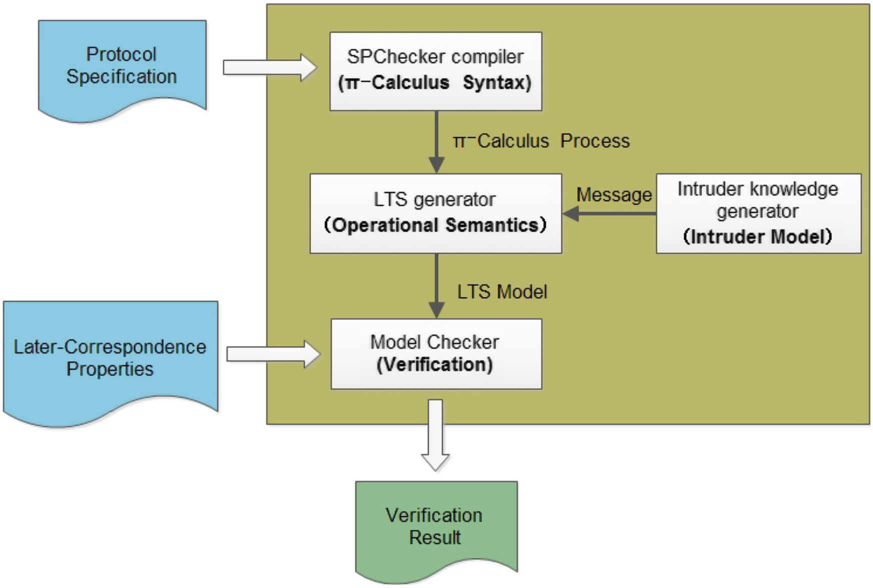


Fig. 1. The architecture

ProVerif has extended the applied π-calculus with destructor application and event. The destructor application let $x = g(M_1, \ldots, M_l)$ in P else Q represents if it evaluates $g(M_1, M_l)$ successfully, the result will be assigned to x and P is executed, else Q is executed. Function application and destructor application can be private or public, the public one will be owned by all agents included the intruders and the private one cant be known by intruders. Using functions and destructors, the data structures and cryptographic operations can be described. $event(M).\ P$ means P will be executed after $event(M)$ executes, it is used to specify correspondence.

3.2 Operational Semantics

The semantics model is based on labeled transition systems, we review the LTS firstly.

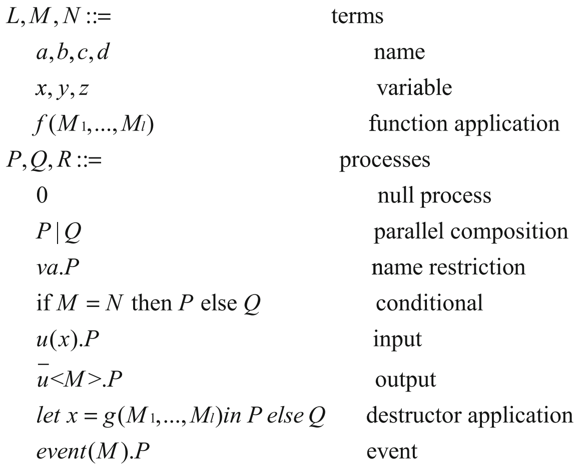
Definition 1 (Labeled Transition System). An LTS is a 3-tuple $L = (S, init, T)$ where S is the state set,$init \in S$ is the initial state and $T : S \times \sum \tau \times S$ is the states transition. Let $\sum \tau$ is a set of all events, $\sum *$ is a set of all traces. Some related relations is defined as follows.

- $s \xrightarrow{e_1,e_2,\ldots,e_n} s'$:For $s', s \in S$, if exits $s_0, ..., s_n \in S$ for all $0 \leq i < n$ such that $S_i \xrightarrow{e_{i+1}} S_{i+1}$ where $e_i \in \sum \tau, s_0 = s$ and $s_n = s'$
- $s \rightarrow^* s'$:If exiting $e_1, ..., e_n \in \sum \tau$ such that $s \xrightarrow{e_1,\ldots,e_n} s'$. In particular, $s \rightarrow^* s$.

- Let $tr : \sum *$ is a sequence of events. $s \overset{tr}{\Longrightarrow} s'$ means if exiting $e_1, ..., e_n \in \sum \tau$ such that $s \xrightarrow{e_1,...,e_n} s'$, then $tr = \langle e_1, ..., e_n \rangle$ is one trace of L.
- $enabled(s) = \{e : \sum \tau | \exists s' \in S, s \xrightarrow{e} s'\}$.
- The set of traces is $traces(L) = \{tr : \sum * \mid \exists s' \in S, init \overset{tr}{\Longrightarrow} s'\}$

A semantic configuration is a 3-tuple(P, V, C), where P is the current process, V is the map of the global variables and local variables which is a set of mappings from a name to a value, and C is a set of channels. Figure 3 presents the rules of operational semantics, $|$ is symmetric and associative.$eval(v, exp)$ evaluates the value of the exp given valuation v. $pair(a, b)$ records a mapping from name a to its value b. The set of bound names $bn(A)$ contains every name n which is under restriction νn inside A. The set of bound variables $bv(A)$ consists of all those variables x occurring in A that are bound by restriction νx or input $u(x)$. The set of free names $fn(A)$ consists of those names n occurring in A not in the scope of the binder νn. The set of free variables $fv(A)$ contains the variables x occurring in A which are not in the scope of a restriction νx or input $u(x)$. τ represents the internal event, we will ignore it when verifying the protocol based on LTS model.

Rule OUT represents when outputting an expression through channel u, it will calculate its value and put it on the channel. The term N will be added in map V if an input term N is input in rule IN. The PAR rule represents the parallel composition of P and Q. In rule RES, a new name a is created and it will record the name a and its value a in map V. $DESTR1$ and $DESTR2$ is the rule of destructor application. It will perform process P and record $pair(x, M')$ if there is a term M' such that $g(M_1, M_2, ..., M_l) = M'$, else it will perform process Q. Rule $COND1$ and $COND2$ mean if M and N reduce to the same



$L, M, N ::=$	terms
a, b, c, d	name
x, y, z	variable
$f(M_1, ..., M_l)$	function application
$P, Q, R ::=$	processes
0	null process
$P \mid Q$	parallel composition
$va.P$	name restriction
if $M = N$ then P else Q	conditional
$u(x).P$	input
$\bar{u}<M>.P$	output
$let\ x = g(M_1, ..., M_l) in\ P\ else\ Q$	destructor application
$event(M).P$	event

Fig. 2. Syntax of the process calculus

$$(\bar{u}<\exp>.P,V,C)\xrightarrow{\bar{u}<eval(v,\exp)>}(P,V,C[u]\cup eval(V,\exp)) \qquad \text{OUT}$$

$$(u(x).P,V,C)\xrightarrow{u(N)}(P,V\cup pair(x,N),C) \qquad \text{IN}$$

$$\frac{(P,V,C)\xrightarrow{a}(P',V',C') \quad bv(a)\cap fv(Q)=bn(a)\cap fn(Q)=\phi}{(P\,|\,Q,V,C)\xrightarrow{a}(P'\,|\,Q,V',C')} \qquad \text{PAR}$$

$$(va.P,V,C)\xrightarrow{va}(P,V\cup pair(a,a),C) \qquad \text{RES}$$

$$\frac{\exists M'\in\sum M, g(M_1,...,M_l)=M'}{(let\, x=g(M_1,...,M_l)\ \ in\, P\, else\, Q,V,C)\xrightarrow{\tau}(P,V\cup pair(x,M'),C)} \qquad \text{DESTR1}$$

$$\frac{\forall M'\in\sum M, g(M_1,...,M_l)\neq M'}{(let\, x=g(M_1,...,M_l)\ \ in\, P\, else\, Q,V,C)\xrightarrow{\tau}(Q,V,C)} \qquad \text{DESTR2}$$

$$\frac{M=N}{(if\ M=N\, then\, P\, else\, Q,V,C)\xrightarrow{\tau}(P,V,C)} \qquad \text{COND1}$$

$$\frac{M\neq N}{(if\ M=N\, then\, P\, else\, Q,V,C)\xrightarrow{\tau}(Q,V,C)} \qquad \text{COND2}$$

$$(event(M).P,V,C)\xrightarrow{event(M)}(P,V,C) \qquad \text{EVENT}$$

Fig. 3. Operator semantics

term at running, it will perform process P, else it will perform process Q. Event rule represents it will execute process P after $event(M)$ executes. The π-calculus can be represented by LTS through the reduction rules.

3.3 Definition of Later-Correspondence

Based on semantics model, we formally define the later-correspondence.Later-correspondence is the form of properties "if the protocol executes e event, then $e_1, ..., e_l$ will be executed later". Like ProVerif the events may include arguments [24], which allows one to relate the values of variables at the various events. The events in our method can also be const terms.

Definition 2. The $event(M)$ will happens in trace $tr = (P_0, V_0, C_0) \rightarrow^* (P', V', C')$ if and only if there exists a reduction$(event(M).P, V, C) \xrightarrow{event(M)} (P, V, C)$.

Definition 3. For a protocol implementation $L = (S, init, T)$, it satisfies the later-correspondence $event(M) \rightarrow \bigwedge_{k=1}^{l} event(M_k)$ if and only if $\forall tr = \langle e_1, ..., e_n\rangle \in traces(L).\exists 1 \leq i \leq n.(e_i = event(M) \bigwedge_{k=1}^{l}(\exists i < j_k \leq n.e_{jk} = event(M_k)))$

3.4 Example

As an example, we model for one Certified Email protocol [29]. As the protocol has some obvious defects in the resolve sub-protocol and we improve it. The result of verification will be analyzed in Sect. 5. The Certified Email consists of two sub-protocols: the exchange protocol and the resolve protocol:

1. Exchange protocol

$$\begin{aligned}
&E1 : A \rightarrow B : h(m), E_{ska}(h(m))\\
&E2 : B \rightarrow A : c, r, s, h(receipt_b), Cert_{tb}, E_{skb}(h(m))\\
&E3 : A \rightarrow B : E_{pkb}(m)\\
&E4 : B \rightarrow A : E_{pka}(receipt_b)
\end{aligned}$$

2. Resolve protocol

$$\begin{aligned}
&R1 : A \rightarrow TPP : Cert_{tb}, E_{skb}(h(m)), E_{pkttp}(m)\\
&R2 : TTP \rightarrow A : E_{pka}(receipt_b)\\
&R3 : TTP \rightarrow B : E_{pkb}(m)
\end{aligned}$$

A, B, TTP represents the sender, receiver and the trusted third party. m is the email they exchange. $receipt_b$ is the receipt of B. It executes the exchange protocol in normal circumstances. When it is abnormal or the dispute exists, resolve protocol will be executed. TTP generates $receipt_b$ for B before protocol executes and B generates the signature (c, r, s) for $receipt_b$. A can check whether it is a valid signature through calculating $(c, r, s) = H(h(receipt_b))$. $Cert_{tb}$ is the certificate that TTP generates for B and $Cert_{tb} = E_{pkttp}(receipt_b)$.

Exchange protocol executes like that: firstly A sends $h(m)$ and the signature of $h(m)E_ska(h(m))$ to B. B will check whether the signature is valid and if it is valid, B sends the signature $(c, r, s), h(receipt_b)$ and $Cert_{tb}$ to A. After receiving the messages, A will check whether the signature (c, r, s) is valid through $h(receipt_b)$ and check the validity of the $Cert_{tb}$.

If the two conditions are satisfied, A makes sure that it can get $receipt_b$ from TTP when it is abnormal. A sends $E_{pkb}(m)$ to B and B will send $E_{pkb}(m)$ to A after checking that m is valid. Resolve protocol executes like that: A sends $Cert_{tb}, E_{skb}(h(m)), E_{pkttp}(m)$ to TTP and TTP will Check the validity of $Cert_{tb}$, restore the $receipt_b$ from $Cert_{tb}$ and check the validity of m. If the two conditions are satisfied, TTP will send the encrypted message $E_{pkb}(m)$ to B and send the encrypted receipt $E_{pka}(receipt_b)$ to A.

This protocol can be formalized by the π-calculus which is introduced in Sect. 3.2. The protocol model includes the process A, process B, process TTP. A initially has message m, the public key and private key of A, the public key of B and TTP. The sk_T in the parameters is used for process TTP,so it is not known by process A.

The agent A can be modeled as follows. $let\ (= crs) = getcrs(qm)\ in \ldots$ This construct is syntactic sugar for $let\ x = getcrs(qm)\ in\ if(x = crs)\ then \ldots$ The last line in the model means the agent A will executes the exchange protocol with TTP.

$$\begin{aligned}
&clienta(pk_A, sk_A, pk_B, pk_T, sk_T) =\\
&\quad \bar{c}\langle(hash(m), aencs(hash(m), sk_A))\rangle.\\
&\quad c(x).\mathbf{let}(crs, qm, zs, bhm) = x\ \mathbf{in}\\
&\quad \mathbf{let}(= crs) = getcrs(qm)\mathbf{in}\\
&\quad \mathbf{if}\ hash(m) = adecs(bhm, pk_B)\ \mathbf{then}\\
&\quad \bar{c}\langle aencsec(m, pk_B)\rangle.c(y)\\
&\quad \bar{c}\langle val(zs, bhm, aencsec(m, pk_T))\rangle | ttp(pk_A, sk_T, pk_B, pk_T)
\end{aligned}$$

B initially has receipt r, the public key and private key of B, the public key of A and TTP, the certificate $Cert_{tb}$. The agent B can be modeled as:

$$\begin{array}{l}
clientb(pk_A, sk_B, pk_B, pk_T, cts) = \\
\quad c(x).\textbf{let}(hm, shm) = x \ \textbf{in}; \\
\quad \textbf{let}(= hm) = adecs(shm, pk_A)\textbf{in} \\
\quad \textbf{let}\ ctsp = adec(cts, sk_B)\textbf{in} \\
\quad \overline{c}\langle(getcrs(hash(r)), hash(r), aenc(ctsp, pk_T), aencs(hm, sk_B))\rangle. \\
\quad c(msg).\textbf{let}\ msgjm = adecsec(msg, sk_B)\textbf{in} \\
\quad \textbf{let}(= hm) = hash(msgjm)\textbf{in}\ \overline{c}\langle aencsec(r, pk_A)\rangle
\end{array}$$

TTP initially has the public key and private key of TTP, the public key of A and B. The agent TTP can be modeled as:

$$\begin{array}{l}
ttp(pk_A, sk_T, pk_B, pk_T) = \\
\quad c(xt).\textbf{let}(ct, ehm, msgpkt) = getval(xt)\textbf{in} \\
\quad \textbf{let}\ cst = adec(ct, sk_T)\textbf{in}\ \textbf{let}\ yt = getrfrombit(cst)\textbf{in} \\
\quad \textbf{let}\ hmt = adecs(ehm, pk_B)\textbf{in}\ \textbf{let}\ msg = adecsec(msgpkt, sk_T)\textbf{in} \\
\quad \textbf{if}\ hash(msg) = hmt\ \textbf{then} \\
\quad \overline{c}\langle aencsec(yt, pk_A)\rangle. \\
\quad \textbf{event}(sendRec(yt)). \\
\quad \overline{c}\langle aencsec(msg, pk_B)\rangle. \\
\quad \textbf{event}(sendMsg(msg))
\end{array}$$

We formalize the later-correspondence "After TTP sends r to A, it will send m to B later" as $\textbf{event}\,(sendRec\,(r)) \rightarrow \textbf{event}\,(sendMsg\,(m))$. In Sect. 4, we will analysis the verification result.

3.5 The Intruder Model

The intruder model considers the malicious behavior of the malicious protocol agents and the intruders. We also regard the malicious agents as intruders. Contrary to honest agents that execute faithfully each statement specified by the protocol, the intruders can intercept, compose, decompose, encrypt, decrypt and send any messages [26]. The message knowledge of intruder is infinite, but most of them are invalid and the valid messages are finite.

The intruder model is based on Dolev-Yao model [26] and it also considers the malicious agent. We simplify the intruder model so that the knowledge is finite. Let $K(M)$ represents the intruder owns message M. F represents the set of fresh values. Pc represents the set of the public channels. t represents term. $SameType(t, M)$ means the types of t and M are the same. $C(u)\backslash t$ means message t is removed from the message set of channel u. Figure 4 gives the intruder semantic and its rules.

The malicious behavior of the protocol agents can also be represented by the intruder behavior. For example, one behavior that agent A sends a invalid message to B can be modeled use the rule *resend*. The intruder blocks the

$t \in F \Rightarrow K(t)$	new
$\bar{u} < t > \wedge u \in Pc \Rightarrow K(t)$	intercept
$K(t) \wedge u \in Pc \Rightarrow \bar{u} < t >$	send
$K(t1) \wedge K(t2) \Rightarrow K((t1,t2))$	compose
$K((t1,t2)) \Rightarrow K(t1) \wedge K(t2)$	decompose
$K(f(M_1,...,M_n)) \wedge \forall 1 \leq i \leq n.(K(t_i) \wedge SameType(t_i, M_i)) \Rightarrow K(f(t_1,...,t_n))$	constructor
$K(g(M_1,...,M_n)) \wedge \forall i.(K(t_i) \wedge SameType(t_i, M_i)) \wedge g(t_1,...,t_n) = t' \Rightarrow K(t')$	deconstructor
$\bar{u} < t > \wedge u \in Pc \wedge K(\mathrm{t'}) \wedge t' \neq t \Rightarrow K(t) \ \wedge C(u) \backslash t \wedge \bar{u} < t' >$	resend

Fig. 4. Intruder rules

valid message t and sends one invalid message t' to B. In the rest we assume the protocol agents are honest and their malicious behavior is represented by intruders.

Because of the state space explosion, we limit the number of protocol sessions and generate the finite effective messages for intruders. SPChecker updates the intruder knowledge through the message pattern. For example from message $aenc(sign((msg, k), skB), pkX)$ in the protocol, we can record the message pattern $aenc(sign((bitstring, key), skey), pkey)$. Figure 5 shows the process of updating the intruder knowledge. Firstly, $SPCheker$ executes the protocol without intruders, we can get the global variables $GlobalMsg$ and messages $PublicChannelMsg$ which are put in the public channel. For every message in $GlobalMsg$, one corresponding attacker message will be generated and both of them are put into $AttackerMsg$. We also can get all message patterns $Message$ $Pattern$ from the messages $PublicChannelMsg$. Secondly, based on the $Message$ $Pattern$ and $AttackerMsg$, all possible messages will be constructed in $ConstructedMsg$. But some messages in $ConstructedMsg$ are invalid. Finally, the invalid messages will be removed through verifying the secrecy by ProVerif. For one message, if its verification of secrecy is true, it will not be owned by the intruder and it is an invalid one. We can make sure that all the messages in the initial intruder knowledge are valid.

3.6 Verification

In this section we will present the important algorithms for verification of the later-correspondence. Given a process P of a security protocol, we will verify the later-correspondence: $event(begin(M)) \rightarrow event(end(M))$. The initial V (this is introduced in the operational semantics) is empty and every channel from set C has no terms. The initial protocol configuration can be (P, V, C). We will get the $LTS(S, init, T)$ based on the semantics rules, where $S = \{s|(P, V, C) \rightarrow^{*} s\}$,$init = (P, V, C)$, $T = \{(s_1, e, s_2) : S \times \sum \tau \times S | s_1 \xrightarrow{e} s_2, s_1 \in S, s_2 \in S\}$.

Form the LTS model $(S, init, T)$, we have all traces of process P. For every path in the trace model, we will check whether "If begin(M) has executed, then end(M) will be executed eventually" is true. As shown in Table 1, we can check

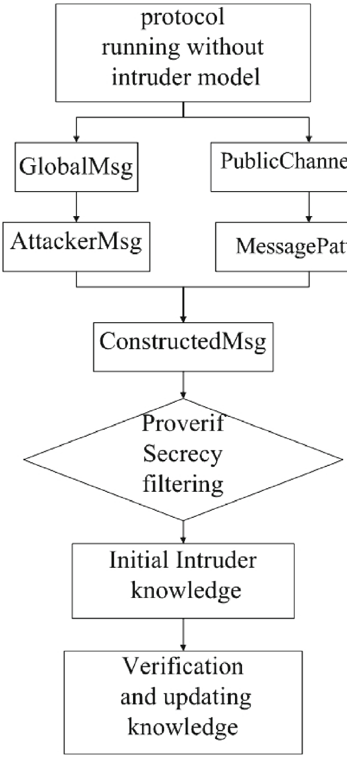

protocol running without intruder model
GlobalMsg
PublicChannelMsg
AttackerMsg
MessagePattern
ConstructedMsg
Proverif Secrecy filtering
Initial Intruder knowledge
Verification and updating knowledge

Fig. 5. Process of constructing intruder messages

the property through the events sequence in the trace. In the procedure, the on-the-fly technique is used and if the proposition of one trace is false, the procedure will stop and we get one counter-example. This technique reduces the verification time greatly.

Termination. In order to avoid states explosion, we limit the number of protocol sessions. The intruder knowledge is also finite through efficient messages generation procedure. So the traces of LTS model from the process P is finite. Because the sequential processes in process P are finite, the events in every trace are also finite. In conclusion, the procedure which checks the proposition in every trace can be terminated in a limited time.

Verification Algorithm. Let *impl* is the initial configuration of the protocol implementation and *property* is the later-correspondence that needs to be verified. Figure 6 shows the later-correspondence verification algorithm. The algorithm checks every trace which is from the LTS model. LTS model will be constructed in algorithm *next*. For simplicity some details of the procedure are skipped and we omit the procedure for producing the counterexample.

```
procedure CorrespondencyVerify(impl, property)
1.  pending.push(impl);
2.  while(pending $\neq \varnothing$)
3.      im = pending.pop();
4.      if (next(im) = $\varnothing$)
5.          if (im.state $\neq 0 \wedge$ im.state $\notin$ mark)
6.              return false;
7.          endif
8.      endif
9.      foreach (im' $\in$ next(im))
10.         im'.state = im.state + mark[im'.event];
11.         if (im'.event $\in$ property.seEvents $\wedge$ im.state $\neq 0 \wedge$ im'.state $\neq 0$)
12.             return false;
13.         endif
14.         pending.push(im');
15.     endfor
16. endwhile
17. return true;
```

Fig. 6. Algorithm: $correspondenceVerify(impl, property)$

The procedure will check the later-correspondence using a depth-first-search way. *pending* is a stack and the initial configuration is put into *pending* in line 1. From line 2 to line 16, it will check for every configuration in the stack until finding one counterexample. The function *next* is presented in Fig. 7. Given a configuration *im*, it will return a set of configurations for each event. *event* in the procedure represents the event through which the configuration can have a reduction. *mark* is the set of weight for events. For every two events ($e1$,$e2$) of the property, SPChecker will generate unique weight for them and their sum is zero. The weight of other events which are not in the property is zero. *state* means the current weight of the configuration and it will be calculated from the last configuration. Line 4 to line 8 means that the current path ends and *im* has no next configurations, so it will check the state of the last configuration. In order to prevent producing the first type of attacks which are introduced in Sect. 1 and they are caused by the interruption of messages, we ignore the counterexamples that only execute the first event. In line 5 if the state is not equal to zero and it does not only execute one event, it is a counterexample. Line 9 to line 15 checks every configuration in the *next* set. When every second-event appears,

```
procedure $next(im)$
$ims = \varnothing$;
foreach $(e \in enabled(im))$
    foreach $im'$ such that $im \xrightarrow{e} im'$
        $ims = ims \cup \{im'\}$;
    endfor
endfor
return $ims$;
```

Fig. 7. Algorithm: $next(im)$

it will check the validity in Line 11. If both states of the last configuration and the current configuration are not equal zero after one second-event, it is an counterexample. In line 14, the current configuration is put into the stack *pending* for checking the rest part.

4 Results Analysis

We have implemented our verifier based on PAT and have verified various authentication protocols. Table 2 shows the experimental results for various protocols on a CPU 3.0 GHz Core Duo CPU and 4 GB RAM.

Using the SPChecker, we verified the later-correspondence of the certified email protocol, the AndrewSecureRPC protocol, the Handshake protocol, the simplified NeedhamSchroeder protocol and the Woo-Lam protocol. We also found counterexamples from all of the protocols. The second column describes the time of constructing initial knowledge and the third column presents the verification time. We can find that the intruder knowledge is constructed in a shorter time. The result shows our strategies discussed can reduce the number of messages effectively. We believe that the performance of our verifier will be improved significantly using more reduction techniques.

We analysis the verification of the certified email protocol here. In order to verify the later-correspondence "TTP sends r to A, and then it will send m to B later". We formalize the property as: $event\,(sendRec\,(r)) \rightarrow event\,(sendMsg\,(m))$.

Table 2. Experimental results

	Intruder knowledge constructed time	Verification time
Certified Email protocol [29]	1.37 s	5 s
AndrewSecureRPC protocol [28]	1.61 s	2 s
NeedhamSchroeder protocol [30]	1.02 s	22 s
Handshake protocol [30]	0.79 s	27 s
Woo-Lam protocol [28]	0.65 s	379 s

1. $init$- >
2. $c!pk(skA)$- >
3. $c!pk(skB)$- >
4. $c!pk(skT)$- >
5. $c?(hash(m), aencs(hash(m), skA))$- >
6. $c!(getcrs(hash(r)), hash(r), aenc(tobitstr(r), pk(skT)), aencs(hash(m), skB))$- >
7. $c?aenc\,\mathrm{sec}(m, pk(skB))$- >
8. $c!aenc\,\mathrm{sec}(r, pk(skA))$- >
9. $c!(hash(m), aencs(hash(m), skA))$- >
10. $c?(getcrs(hash(r)), hash(r), aenc(tobitstr(r), pk(skT)), aencs(hash(m), skB))$- >
11. $c!aenc\,\mathrm{sec}(m, pk(skB))$- >
12. $c?aenc\,\mathrm{sec}(m, pk(skB))$- >
13. $c?val(aenc(tobitstr(r), pk(skT)), aencs(hash(m\_Attac\,\mathrm{ker}), skB), aenc\,\mathrm{sec}(m\_Attac\,\mathrm{ker}, pk(skT)))$- >
14. $c!aenc\,\mathrm{sec}(r, pk(skA))$- >
15. $start\_send\,\mathrm{Re}\,c(r)$- >
16. $c!aenc\,\mathrm{sec}(m\_Attac\,\mathrm{ker}, pk(skB))$- >
17. $end\_sendMsg(m\_Attac\,\mathrm{ker})$- >
18. $c!val(aenc(tobitstr(r), pk(skT)), aencs(hash(m), skB), aenc\,\mathrm{sec}(m, pk(skT)))$- >

Fig. 8. The counter-example

In the Fig. 8, it represents the counterexample. Because of the different order of executing in parallel process,the sequence of the events in the counterexample are not exactly same with the protocol performing sequence. TTP receives message $c?val(aenc(tobitstr(r), pk(skT)), aencs(hash(m\_Attacker), skB), aencsec$ $(m\_Attacker, pk(skT)))$ in line 13. It makes that TTP sends $aencsec(r, pk(skA))$ to A in line 14 and sends $aencsec(m\_Attacker, pk(skB))$ in line 16.

The counterexample shows that after event $sendRec(r)$ executes, it executes the event $sendMsg(m\_Attacker)$ rather than $sendMsg(m)$. So the later-correspondence is not satisfied. Because of the reply attack, the intruder (intruder or malicious agent) recorded the message about $m\_Attacker$ from the previous run and sends it to TTP in the next run. At last, the agent A obtains the receipt r but B can't get message m.

5 Conclusion and Future Works

This paper has given the definition of later-correspondence. Later-correspondence describes the expected order of messages which has been sent and received. Using the input language of ProVerif [8], we presented an automatic method for verifying later-correspondence. An automatic verifier SPChecker has been developed based on PAT [23]. Not like other model checking tools, SPChecker can construct intruder model through ProVerif filtering automatically. In order to solve the states explosion problem, we limit the number of sessions and took some strategies, the experimental results have demonstrated its effectiveness. The counterexample of the Certified Email protocol shows that the protocol does't satisfy the later-correspondence because of the existing replay attack.

For the future work, we will improve our intruder model and apply more reduction techniques to improve the performance. Secondly, we will extend SPChecker and make it verify the non-repudiation and fairness based on correspondence in e-commerce protocols.

Acknowledgments. This work was supported in part by the National Science Foundation of China (No. 91118003, 61272106, 61003080) and 985 funds of Tianjin University.

References

1. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. Proc. R. Soc. Lond. A **426**, 233–271 (1989)
2. Lowe, G.: Breaking and fixing the needham-schroeder public-key protocol using FDR. In: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems, pp. 147–166 (1989)
3. Xie, X.F., Li, X.H., Cao, K.Y., Feng, Z.Y.: Security modeling based on CSP for network protocol. Int. J. Digit. Content Technol. Appl. **6**, 496–504 (2012)
4. Thayer, F.J., Herzog, J.C., Guttman, J.D.: Strand spaces: why is a security protocol correct? In: Proceedings of the 1998 IEEE Symposium on Security and Privacy, pp. 160–171 (1998)
5. Thayer, F.J., Herzog, J.C., Guttman, J.D.: Strand spaces: proving security protocols correct. J. Comput. Secur. **7**, 191–230 (1999)
6. Bella, G., Paulson, L.C.: Using Isabelle to prove properties of the Kerberos authentication system. In: DIMACS Workshop on Design and Formal Verification of Security Protocols (1997)
7. Athena, D.X.S.: A New efficient automatic checker for security protocol analysis. In: Computer Security Foundations Workshop, pp. 192–202 (1999)
8. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In:Computer Security Foundations Workshop, pp. 82–96 (2001)
9. Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Drielsma, P.H., Heám, P.C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santiago, J., Turuani, M., Viganò, L., Vigneron, L.: The AVISPA tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
10. Meadows, C.: The NRL protocol analyzer: an overview. J. Logic Program. **26**, 113–131 (1996)
11. Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: introducing a process analysis toolkit. In: Margaria, T., Steffen, B. (eds.) ISoLA 2008. CCIS, vol. 17, pp. 307–322. Springer, Heidelberg (2008)
12. Mitchell, J.C., Mitchell, M., Stern, U.: Automated analysis of cryptographic protocols using Murphi. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 141–151. IEEE Computer Society Press (1997)
13. Hoare, C.: Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall, Upper Saddle River (1985)
14. Gordon, A., Jeffrey, A.: Authenticity by typing for security protocols. J. Comput. Secur. **11**, 451–519 (2003)

15. Gordon, A., Jeffrey, A.: Types and effects for asymmetric cryptographic protocols. J. Comput. Secur. **12**, 435–484 (2004)
16. Gordon, A.D., Hüttel, H., Hansen, R.R.: Type inference for correspondence types. In: 6th International Workshop on Security Issues in Concurrency (2008)
17. Bugliesi, M., Focardi, R., Maffei, M.: Analysis of typed analyses of authentication protocols. In: Proceedings 18th IEEE Computer Security Foundations Workshop, pp. 112–125 (2005)
18. Bugliesi, M., Focardi, R., Maffei, M.: Dynamic types for authentication. J. Comput. Secur. **15**, 563–617 (2007)
19. Cremers, C., Mauw, S., de Vink, E.: Defining authentication in a trace model. In: Proceedings of the First International Workshop on Formal Aspects in Security and Trust, pp. 131–145 (2003)
20. Corin, R., Saptawijaya, A., Etalle, S.: A logic for constraint based security protocol analysis. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 155–168 (2006)
21. Schmidt, B., Meier, S., Cremers, C., Basin, D.: Automated analysis of Difie-Hellman protocols and advanced security properties. In: Computer Security Foundations Symposium (CSF), pp. 78–94 (2012)
22. Tuan, L.A., Sun, J., Liu, Y., Dong, J.S., Li, X.H., Tho, Q.T.: SEVE: automatic tool for verification of security protocols. Front. Comput. Sci. Spec. Issue Form. Eng. Method **6**, 57–75 (2012)
23. Liu, Y., Sun, J., Dong, J.S.: Developing model checkers using PAT. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 371–377. Springer, Heidelberg (2010)
24. Blanchet, B.: Automatic verification of correspondences for security protocols. J. Comput. Secur. **17**, 363–434 (2009)
25. Woo, T.Y.C., Lam, S.S.: A semantic model for authentication protocols. In: IEEE Symposium on Security and Privacy, pp. 178–194 (1993)
26. Dolev, D., Yao, A.C.: On the security of public-key protocols. IEEE Trans. Inf. Theory **2**, 198–208 (1983)
27. Ryan, M.D., Smyth, B.: Applied pi calculus. In: Cortier, V., Kremer, S. (eds.) Formal Models and Techniques for Analyzing Security Protocols. IOS Press, Amsterdam (2011)
28. Clark, J.A., Jacob, J.L.: A survey of authentication protocol literature: version 1.0 (1997). http://www.cs.york.ac.uk/jac/papers/drareview.ps.gz
29. Zhang, Q., Zhang, L., et al.: A new certified E-mail protocol based on signcrytion. J. Univ. Electron. Sci. Technol. China **37**, 282–284 (2008)
30. Blanchet, B., Smyth, B.: ProVerif 1.86pl3: automatic cryptographic protocol verifier, user manual and tutorial (2011). http://prosecco.gforge.inria.fr/personal/bblanche/proverif/manual.pdf

Combining Separation Logic and Projection Temporal Logic to Reason About Non-blocking Concurrency

Xiaoxiao Yang(✉)

State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, Beijing 100190, China
xxyang@ios.ac.cn

Abstract. Projection temporal logic programming language MSVL can be well used in modelling simulation and verification of concurrent programs. Non-blocking programs have been widely used in multiprocessor systems. In this paper, we combine separation logic and projection temporal logic to reason about non-blocking concurrency. To this end, we use separation logic as state assertions and projection temporal logic as interval assertions to specify the spatial and temporal properties. Then we extend the MSVL language with atomic blocks, the pointer assignment and the interleaving operator to simulate various of non-blocking programs. Further, we give a sound axiomatic system for the new extended MSVL and prove the lock-free property of Treiber's stack using the axiomatic system.

1 Introduction

Parallelism has become a challenging domain for processor architectures, programming, and formal methods [1]. Based on the multiprocessor systems, threads communicate via shared memory, which employs some kind of synchronization mechanisms to coordinate access to a shared memory location. Verification of such synchronization mechanisms is the key to assure the correctness of shared-memory programs.

There are two main forms of synchronization in software systems: one is blocking synchronization and the other is non-blocking synchronization. Blocking synchronization uses mutual exclusion locks to protect a data structure such that only one thread may access the data structure at any time. This approach severely limits parallelism, negating some of the benefits of multicore and multiprocessor systems, Nowadays, nonblocking synchronization has become an important basis for multiprocessor programming. However, nonblocking concurrency is rather complicated and error-prone. Its correctness is usually far from obvious and is hard to verify. Deciding how to efficiently verify nonblocking concurrency in multicore systems has received more and more attention from both academic and industrial communities.

This research was supported by NSFC 61100063 and by a Humboldt Fellowship (X.Y.) from Alexander von Humboldt Foundation.

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 127–144, 2015.
DOI: 10.1007/978-3-319-17404-4_9

However, most existing verification techniques such as invariants [2], reduction [3,4] and ownership [5,6] do not enable to deal with nonblocking concurrency. More advanced techniques such as simulation [7,8] are too general and complicated to readily verify nonblocking programs. Rely-guarantee compositional method is an efficient verification technique, which has been widely used in concurrency [9–12]. However, different from the thought of modular method, we will do the verification in a unified framework such that programs and properties can be written in one notation. This can reduce the burden of verification that transform between different formal notations and to some extent improve the efficiency.

Temporal logic has proved very useful in specifying and verifying concurrent programs [13] and has seen particular success in the temporal logic programming model, where both algorithms and their properties can be specified in the same language [14]. Indeed, a number of temporal logic programming languages have been developed for the purpose of simulation and verification of software and hardware systems, such as Temporal Logic of Actions (TLA) [15], Cactus [16], Tempura [17], MSVL [18,19], etc. All these make a good foundation for applying temporal logic to implement and verify concurrent algorithms. Particularly, Modeling Simulation and Verification Language (MSVL) is an executable subset of projection temporal logic (PTL) [20,21]. It could facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way. Furthermore, in practice, the framing operators and minimal model semantics in MSVL enable us to narrow the gap between temporal logic and programming languages in a realistic way.

As a specification language, separation logic (SL) [22] is an extension of Hoare logic [23], where the program state in a precondition and postcondition consists of a stack and a heap. Its main application so far has been the reasoning about pointer programs that keep track of the memory. SL provides a series of useful concepts such as the separating conjunction and the frame rule to support local reasoning of programs. In addition, O'Hearn has proposed Concurrent Separation Logic (CSL) [24] by using invariants to reason about concurrent pointer programs.

Therefore, in the paper, we are motivated to extend the projection temporal logic with the thought of resource separation in separation logic to specify various temporal properties in nonblocking concurrency with dynamic data structures with pointers. To formalize the non-blocking algorithms, we introduce the pointer assignment, atomic blocks and the interleaving operator into MSVL [25, 26]. Atomic blocks can be used to model architecture-supported atomic instructions, such as compare-and-swap (CAS), or to model high-level transactions implemented by software transactional memory (STM). Interleaving operator enforces that access to shared memory locations must be done in atomic blocks, otherwise the formulas can result in false. We define the pointer operators like $\&x$ and $*Z_v$ and the atomic blocks as the primitive terms and formulas in PTL. The interleaving operator can be directly derived from the PTL basic formulas. Furthermore, in order to verify non-blocking programs in a unified framework, we augment the axiomatic system of MSVL [27] with new axioms ($e.g.$ $A17 - A22$, $Atom$) and the inference rules ($e.g.$ $Rule - Atom$, $Rule - PointerI$, $Rule - PointerII$), and prove that the new extended axiomatic system is sound. Moreover, we take the Treiber's stack as an example to illustrate that the new extended MSVL can be well used to formalize the non-blocking operations pop and push. Finally, we prove that the Treiber's stack satisfies the lock-free property within the axiomatic system.

The paper is organized as follows: in Sect. 2, we define an interval-based assertion language that combines separation logic and projection temporal logic. We also define the atomic blocks and the pointer assignment as the basics of PTL. In Sect. 3, we give an extended MSVL and a sound axiomatic system with new axioms and inference rules. In Sect. 4, an example is presented to illustrate how the MSVL and the axiomatic system can be used to specify and verify the non-blocking concurrency. Section 5 draws the conclusion.

2 Interval-Based Assertion Language

2.1 Syntax and Semantics

The assertion language we use includes state assertions and interval assertions. We employ a subset of separation logic as state assertions for specifying shared mutable data structures. State assertions is defined as follows.

$$Q ::= \mathsf{emp}_h \mid \mathsf{emp}_s \mid E_1 \mapsto E_2 \mid Q_1 \odot Q_2$$

where emp_h denotes an empty heap and emp_s an empty stack, $E_1 \mapsto E_2$ is a singleton heap, where E_1, E_2 are expressions (i.e., terms), $Q_1 \odot Q_2$ is the separating conjunction.

Figure 1 defines the program states and intervals, where a state is a pair of stack and heap, i.e., $\delta = (s, h)$, an interval is a finite or infinite sequence of states. Let dom denote the domain of heap and stack, which is defined as $\mathsf{dom} : \mathsf{heap} \to 2^{\mathsf{Nat}}$ or $\mathsf{dom} : \mathsf{stack} \to 2^{\mathsf{Var}}$. The semantics of the selected separation logic assertions is given in Fig. 2, where $\Im[E]$ means the evaluation of expression E.

$$\begin{array}{ll}
\mathsf{stack}: & s \in \mathsf{Var} \to \mathsf{Int} \\
\mathsf{heap}: & h \in \mathsf{Nat} \to \mathsf{Int} \\
\mathsf{state}: & \delta \in \mathsf{stack} \times \mathsf{heap} \\
\mathsf{interval}: & \sigma = \langle \delta_0, \delta_1, \ldots \rangle \\
f \perp g & \overset{\text{def}}{=} \mathsf{dom}(f) \cap \mathsf{dom}(g) = \emptyset \\
(s,h) \uplus (s',h') & \overset{\text{def}}{=} (s \cup s', h \cup h'),\ \text{if } s \perp s', h \perp h'
\end{array}$$

Fig. 1. Program states and intervals

$$\begin{array}{ll}
(s,h) \models_{\mathsf{SL}} \mathsf{emp}_s & \text{iff } s = \emptyset \\
(s,h) \models_{\mathsf{SL}} \mathsf{emp}_h & \text{iff } h = \emptyset \\
(s,h) \models_{\mathsf{SL}} E_1 \mapsto E_2 & \text{iff there exist } l \text{ and } n \text{ such that} \\
 & \quad \Im[E_1] = l, \Im[E_2] = n, \mathsf{dom}(h) = \{l\} \text{ and } h(l) = n \\
\delta \models_{\mathsf{SL}} Q_1 \odot Q_2 & \text{iff there exixst } \delta_1 \text{ and } \delta_2 \\
 & \quad \text{such that } \delta_1 \uplus \delta_2 = \delta, \delta_1 \models_{\mathsf{SL}} Q_1 \text{ and } \delta_2 \models_{\mathsf{SL}} Q_2
\end{array}$$

Fig. 2. Semantics for state assertions.

In the following, we discuss the interval assertions $\alpha\mathsf{PTL}$, which augments PTL with the reference operator (&) and the dereference operator ($*$) and atomic blocks ($\langle\rangle$). We follow the thought of [26] that gives the formal definitions of reference and dereference operations based on names. However, for clarity, we have some conventions of variables. Let $\mathsf{Var} = \{x, y, z \ldots\}$ be a set of variables that is defined as $\mathsf{Var} \overset{\text{def}}{=} \mathsf{IntVar} \cup \mathsf{PVar}$, where IntVar is a set of integer variables denoted by $sx, sy, sz \ldots$, of which the values are integers, and PVar is a set of pointer variables $X, Y, Z, \ldots$, of which the values are addresses. Further, $\mathsf{Pointers}$ consists of PVar and PConst, where PConst denotes pointer constants like the form of $\&x$. We use $X_v, Y_v, Z_v \ldots$ to denote $\mathsf{Pointers}$. In this paper, we regard variable names as the addresses of memory cells. We therefore define function $\gamma : \mathsf{PVar} \to \mathsf{Var}$, which is a mapping from pointer variables to addresses.

$\alpha\mathsf{PTL}$ terms and formulas are defined by the following grammar:

$$\begin{array}{lll}
\alpha\mathsf{PTL}\text{ terms:} & e, e_1, \ldots, e_m & ::= sx \mid Z \mid \&x \mid *Z_v \mid f(e_1, \ldots, e_m) \mid \bigcirc e \mid \ominus e \\
\alpha\mathsf{PTL}\text{ formulas:} & p, q, p_1, p_m & ::= \pi \mid e_1 = e_2 \mid \mathrm{Pred}(e_1, \ldots, e_m) \mid \neg p \mid p \wedge q \mid Q \mid \langle p \rangle \\
 & & \quad \bigcirc p \mid (p_1, \ldots, p_m) prj\ q \mid \exists x.p \mid p^{+}
\end{array}$$

where sx is an integer variable, Z is a pointer variable, $\&x$ is a reference operation and $*Z_v$ is a dereference operation. As usual, f ranges over a predefined set of function symbols, $\bigcirc e$ and $\ominus e$ indicate that term e is evaluated on the next and previous states respectively; π ranges over a predefined set of atomic propositions, and $\mathrm{Pred}(e_1, \ldots, e_m)$ represents a predefined predicate constructed with $e_1, \ldots, e_m$; operators *next*($\bigcirc$), *projection operator* (prj) and *chop plus* ($^{+}$) are temporal operators; Q is state assertions defined in Fig. 2 and $\langle p \rangle$ denotes atomic blocks.

An *interval* $\sigma = \langle \delta_0, \delta_1, \ldots \rangle$ is a non-empty sequence of states. An *interpretation* over intervals is $\mathfrak{I} = (\sigma, i, j)$, where i is non-negative integers, and j is an integer or ω, such that $i \le j \le |\sigma|$. We use (σ, i, j) to mean that a formula is interpreted over a subinterval $\sigma_{(i..j)}$. Especially, $\theta_i = (s_i, \gamma_i)$ denotes the i^{th} state. The length of σ, denoted by $|\sigma|$, is defined as follows.

$$|\sigma| = \begin{cases} n & \text{if } \sigma = \langle \delta_0, \ldots \delta_n \rangle \\ \omega & \text{if } \sigma \text{ is infinite} \end{cases}$$

The semantics of $\alpha\mathsf{PTL}$ terms and formulas are shown in Figs. 3 and 4 respectively. For a variable x, we will denote $\sigma' \overset{x}{=} \sigma$ whenever σ' is an interval which is the same as σ except that different values can be assigned to x, and we call σ and σ' are x-equivalent. Other structures such as projection (prj) and atomic blocks ($\langle\rangle$) are well defined in [20, 25]. In the following, we will focus on explaining the definition of atomic blocks in αPTL.

2.2 Framing Issue and Atomic Blocks

Framing concerns how the value of a variable can be carried from one state to the next. Within ITL community, Duan and Maciej [20] proposed a framing technique through an explicit operator ($\mathsf{frame}(x)$), which enables us to establish

$$
\begin{array}{ll}
\Im[sx] & = s_i[sx] = \Im_v^i[sx] \\
\Im[Z] & = \gamma_i(Z),\ \text{where } Z \in \mathsf{PVar} \\
\Im[\&x] & = x,\ \text{where } x \in \mathsf{Var} \\
\Im[*Z_v] & = \Im[\Im[Z_v]],\ \text{where } Z_v \in \mathsf{Pointers} \\
\Im[f(e_1,\ldots,e_m)] & = \begin{cases} f(\Im[e_1],\ldots,\Im[e_m]) & \text{if } \Im[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases} \\
\Im[\bigcirc e] & = \begin{cases} (\sigma, i+1, j)[e] & \text{if } i < j \\ nil & \text{otherwise} \end{cases} \\
\Im[\circleddash e] & = \begin{cases} (\sigma, i-1, j)[e] & \text{if } 0 < i \\ nil & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 3. Interpretation of αPTL terms.

$$
\begin{array}{lll}
(\sigma,i,j) \models \pi & \text{iff} & s_i[\pi] = \mathsf{True} \\
(\sigma,i,j) \models e_1 = e_2 & \text{iff} & (\sigma,i,j)[e_1] = (\sigma,i,j)[e_2] \\
(\sigma,i,j) \models \mathrm{Pred}(e_1,\ldots,e_m) & \text{iff} & \mathrm{Pred}((\sigma,i,j)[e_1],\ldots,(\sigma,i,j)[e_m]) = \mathsf{True} \\
(\sigma,i,j) \models \neg p & \text{iff} & (\sigma,i,j) \not\models p \\
(\sigma,i,j) \models p \wedge q & \text{iff} & (\sigma,i,j) \models p \text{ and } (\sigma,i,j) \models q \\
(\sigma,i,j) \models Q & \text{iff} & \delta_i \models_{\mathsf{SL}} Q \\
(\sigma,i,j) \models \langle p \rangle & \text{iff} & \text{there exists a finite interval } \sigma' \text{ and } I'_{prop} \text{ such that} \\
 & & \langle s_i \rangle \cdot \sigma' \cdot \langle s_{i+1}|_{I'_{prop}} \rangle \models p \wedge \mathsf{FRM}(V_p) \\
(\sigma,i,j) \models \bigcirc p & \text{iff} & i < j \text{ and } (\sigma, i+1, j) \models p \\
(\sigma,i,j) \models \exists x.p & \text{iff} & \text{there exists } \sigma' \text{ such that } \sigma' \overset{x}{=} \sigma \text{ and } (\sigma',i,j) \models p \\
(\sigma,i,j) \models (p_1,\ldots,p_m)\,\mathsf{prj}\ q & \text{iff} & \text{if there are } i = r_0 \leq r_1 \leq \ldots \leq r_m \leq j \text{ such that} \\
 & & (\sigma, r_0, r_1) \models p_1 \text{ and } (\sigma, r_{l-1}, r_l) \models p_l \text{ for all } 1 < l \leq m \text{ and} \\
 & & (\sigma', 0, |\sigma'|) \models q \text{ for } \sigma' \text{ given by :} \\
 & & (1)\ r_m < j \text{ and } \sigma' = \sigma \downarrow (r_0,\ldots,r_m) \cdot \sigma_{(r_m+1..j)} \\
 & & (2)\ r_m = j \text{ and } \sigma' = \sigma \downarrow (r_0,\ldots,r_h) \text{ for some } 0 \leq h \leq m. \\
(\sigma,i,j) \models p^+ & \text{iff} & \text{if there are } i = r_0 \leq r_1 \leq \ldots \leq r_{n-1} \leq r_n = j\ (n \geq 1) \text{ such that} \\
 & & \qquad (\sigma, r_0, r_1) \models p \text{ and } (\sigma, r_{l-1}, r_l) \models p\ (1 < l \leq n)
\end{array}
$$

Fig. 4. Semantics for αPTL formulas.

a flexible framed environment where framed and non-framed variables can be mixed, with frame operators being used in sequential, conjunctive and parallel manner, and an executable version of framed temporal logic programming language is developed [18,19]. The key characteristic of the frame operator can be stated as: $\mathsf{frame}(x)$ *means that variable x keeps its old value over an interval if no assignment to x has been encountered.*

The framing technique defines a primitive proposition p_x for each variable x: intuitively p_x denotes an assignment of a new value to x—whenever such an assignment occurs, p_x must be true; however, if there is no assignment to x, p_x is unspecified, and in this case, we will use a *minimal model* [18,20] to force it to be false. We also call p_x an *assignment flag*. Formally, $\mathsf{frame}(x)$ is defined as follows:

$$\mathsf{frame}(x) \overset{\text{def}}{=} \Box(\mathsf{more} \rightarrow \bigcirc\, \mathsf{lbf}(x)),\ \text{where } \mathsf{lbf}(x) \overset{\text{def}}{=} \neg p_x \rightarrow \exists\, b\ :\ (\ominus x = b \wedge x = b)$$

Intuitively, $\mathsf{lbf}(x)$ (looking back framing) means that, when a variable is framed at a state, its value remains unchanged (same as at the previous state) if no assignment occurs at that state. We say a program is framed if it contains $\mathsf{lbf}(x)$ or $\mathsf{frame}(x)$.

The essence of atomic execution is twofold: first, the concrete execution of the code block inside an atomic wrapper can take multiple state transitions; second, nobody out of the atomic block can see the internal states. This leads to an interpretation of atomic interval formulas based on two levels of intervals—at the outer level an atomic interval formula $\langle p \rangle$ always specify a single transition between two consecutive states, which the formula p will be interpreted at another interval (the inner level), which we call an *atomic interval* and must be finite, with only the first and final states being exported to the outer level. The key point of such a two-level interval based interpretation is the exportation of values which are computed inside atomic blocks. We shall show how framing technique helps at this aspect. A few notations are introduced to support formalizing the semantics of atomic interval formulas.

- Given a formula p, let V_p be the set of free variables of p. we define formula $\mathsf{FRM}(V_p)$ as follows:

$$\mathsf{FRM}(V_p) \overset{\text{def}}{=} \begin{cases} \bigwedge_{x \in V_p} \mathsf{frame}(x) & \text{if } V_p \neq \emptyset \\ \mathsf{True} & \text{otherwise} \end{cases}$$

  $\mathsf{FRM}(V_p)$ says that each variable in the set V_p is a framing variable that allows to inherit the old value from previous states. $\mathsf{FRM}(V_p)$ is essentially used to apply the framing technique within atomic blocks, and allows values to be carried throughout an atomic block to its final state, which will be exported.
- Interval concatenation is defined by

$$\sigma \cdot \sigma' = \begin{cases} \sigma & \text{if } |\sigma| = \omega \text{ or } \sigma' = \epsilon \\ \sigma' & \text{if } \sigma = \epsilon \\ \langle s_0, ..., s_i, s_{i+1}, ... \rangle & \text{if } \sigma = \langle s_0, ..., s_i \rangle \text{ and } \sigma' = \langle s_{i+1}, ... \rangle \end{cases}$$

- If $s = (I_{var}, I_{prop})$ is a state, we write $s|_{I'_{prop}}$ for the state (I_{var}, I'_{prop}), which has the same interpretation for normal variables as s but a different interpretation I'_{prop} for propositions.

In Fig. 5, we present some useful $\alpha\mathsf{PTL}$ formulas that are frequently used in the rest of the paper. ε specifies intervals whose current state is the final state; an interval satisfying more requires that the current state not be the final state; The semantics of $p_1\ ;\ p_2$ says that computation p_2 follows p_1, and the intervals for p_1 and p_2 share a common state. Note that *chop* (;) formula can be defined directly by the projection operator. $\Diamond p$ says that p holds eventually in the future; $\Box p$ means p holds at every state after (including) the current state; $\mathsf{len}(n)$ means that the distance from the current state to the final state is n; skip

specifies intervals with the length 1. $x := e$ means that at the next state $x = e$ holds and the length of the interval over which the assignment takes place is 1. $p^{\circledast}$ means either chop plus p^{+} or ε. Further, $p \equiv q$ (*strong equivalence*) means p and q have the same truth value in all states of every model, whereas $p \supset q$ (*strong implication*) refers to $p \rightarrow q$ always holds in all states of every model.

$$\varepsilon \overset{\text{def}}{=} \neg \bigcirc \mathsf{True} \quad \mathsf{more} \overset{\text{def}}{=} \neg\varepsilon \quad p_1 \mathbin{;} p_2 \overset{\text{def}}{=} (p_1, p_2)\ \mathsf{prj}\ \varepsilon \quad \Diamond p \overset{\text{def}}{=} \mathsf{True} \mathbin{;} p \quad \Box p \overset{\text{def}}{=} \neg\Diamond\neg p$$

$$\mathsf{len}(n) \overset{\text{def}}{=} \begin{cases} \varepsilon & \text{if } n = 0 \\ \bigcirc\mathsf{len}(n-1) & \text{if } n > 1 \end{cases} \qquad \mathsf{skip} \overset{\text{def}}{=} \mathsf{len}(1) \qquad x := e \overset{\text{def}}{=} \bigcirc x = e \wedge \mathsf{skip}$$

$$p^{\circledast} \overset{\text{def}}{=} p^{+} \vee \varepsilon \qquad p \equiv q \overset{\text{def}}{=} \Box(p \leftrightarrow q) \qquad p \supset q \overset{\text{def}}{=} \Box(p \rightarrow q)$$

Fig. 5. Abbreviations for αPTL formulas.

Theorem 1. *The following logic laws are valid.*

*Law*1	$\bigcirc p \supset \mathit{more}$	*Law*2	$\bigcirc(p \wedge q) \equiv \bigcirc p \wedge \bigcirc q$
*Law*3	$\bigcirc(p \vee q) \equiv \bigcirc p \vee \bigcirc q$	*Law*4	$\bigcirc(\exists x : p) \equiv \exists x : \bigcirc p$
*Law*5	$\Box p \wedge \varepsilon \equiv p \wedge \varepsilon$	*Law*6	$\Box p \wedge \mathit{more} \equiv p \wedge \bigcirc\Box p$
*Law*7	$w \mathbin{;} (p \vee q) \equiv (w \mathbin{;} p) \vee (w \mathbin{;} q)$	*Law*8	$(p \vee q) \mathbin{;} w \equiv (p \mathbin{;} w) \vee (q \mathbin{;} w)$
*Law*9	$(w \wedge p) \mathbin{;} q \equiv w \wedge (p \mathbin{;} q)$	*Law*10	$\bigcirc p \mathbin{;} q \equiv \bigcirc(p \mathbin{;} q)$
*Law*11	$\varepsilon \mathbin{;} q \equiv q$	*Law*12	$\exists x : p(x) \equiv \exists y : p(y)$
*Law*13	$\exists x : (p(x) \vee q(x)) \equiv \exists x : p(x) \vee \exists x : q(x)$	*Law*14	$p^{\circledast} \mathbin{;} p \supset p^{+}$
*Law*15	$\varepsilon \equiv \mathit{emp}_h \wedge \mathit{emp}_s$	*Law*16	$\varepsilon\ \mathsf{prj}\ q \equiv q$
*Law*17	$p_1 \wedge \langle p \rangle \equiv p_2 \wedge \langle p \rangle,\ \mathit{if}\ p_1 \equiv p_2$	*Law*18	$\langle p_1 \vee p_2 \rangle \equiv \langle p_1 \rangle \vee \langle p_2 \rangle$

Proof. The proof can be found in [25].

3 Modeling Simulation and Verification Language MSVL

3.1 Extended MSVL for Non-blocking Concurrency

MSVL is used for modeling simulation and verification of hardware and software systems. To describe non-blocking programs, we extend MSVL with the pointer operators, atomic blocks and interleaving operator. The pointer operators and atomic blocks are defined as primitive terms and formulas in αPTL, whereas interleaving operator can be derived from the basics of αPTL. In the following, we present expressions and statements in the extended MSVL.

Expressions. αPTL provides permissible arithmetic expressions and boolean expressions and both are basic terms of αPTL.

Arithmetic expressions:	$e ::= n \mid x \mid \&x \mid *Z_v \mid \bigcirc x \mid \ominus x \mid e_0\ \mathsf{op}\ e_1$
Boolean expressions:	$b ::= \mathsf{True} \mid \mathsf{False} \mid \neg b \mid b_0 \wedge b_1 \mid e_0 = e_1 \mid e_0 < e_1$

where n is an integer, x is a program variable including the integer variables and pointer variables, $\&x$ and $*Z_v$ are pointer operations, op represents common arithmetic operations, $\bigcirc x$ and $\ominus x$ mean that x is evaluated over the next and previous state respectively.

Statements. Figure 6 shows the statements of αPTL, where $p, q, \ldots$ are αPTL formulas. ε means termination on the current state; $x = e$ represents unification over the current state or boolean conditions; $x \Leftarrow e$, $\mathsf{lbf}(x)$ and $\mathsf{frame}(x)$ support framing mechanism; the assignment $x := e$ is as defined in Fig. 5;

Termination :	ε	Unification :	$x = e$
Positive unification :	$x \Leftarrow e$	Unit Assignment :	$x := e$
State frame :	$\mathsf{lbf}(x)$	Interval frame :	$\mathsf{frame}(x)$
Conjuction statement :	$p \wedge q$	Selection statement:	$p \vee q$
Next statement :	$\bigcirc p$	Sequential statement :	$p \,;\, q$
Conditional statement :	$\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q$	Existential quantification :	$\exists x : p(x)$
While statement :	$\texttt{while}\ b\ \texttt{do}\ p$	Atomic block :	$\langle p \rangle$
Parallel statement :	$p \mathbin{\vert\vert\vert} q$	Pointer assignment :	$*Z_v = e$

Fig. 6. Statements in αPTL.

$p \wedge q$ means that the processes p and q are executed concurrently and they share all the states and variables during the execution; $p \vee q$ represents selection statements; $\bigcirc p$ means that p holds at the next state; $p \,;\, q$ means that p holds at every state from the current one till some state in the future and from that state on p holds. $p \wedge q$, $p \vee q$, $\bigcirc p$, $p \,;\, q$ have the straightforward meaning as in the logic. The conditional and while statements are defined as below:

$$\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q \overset{\text{def}}{=} (b \wedge p) \vee (\neg b \wedge q), \qquad \texttt{while}\ b\ \texttt{do}\ p \overset{\text{def}}{=} (p \wedge b)^* \wedge \Box(\varepsilon \rightarrow \neg b)$$

We use a renaming method [19] to eliminate the existential quantification for $\exists x : p(x)$. Pointer assignment $*Z_v = e$ means that the value of e will be assigned to the value (i.e., address) of the pointer variable Z.

$p \mathbin{\vert\vert\vert} q$ executes programs p and q in parallel and we distinguish it from the standard concurrent programs by defining a novel interleaving semantics which tracks only the interleaving between atomic blocks. Intuitively, p and q must be executed at independent processors or computing units, and when neither of them contains atomic blocks, the program can immediately reduce to $p \wedge q$, which indicates that the two programs are executed in a truly concurrent manner. The formal definition of the interleaving semantics will be given in [25]. However, the new interpretation of parallel operator will force programs running at different processors to execute synchronously as if there is a global clock, which is certainly not true in reality. For instance, consider the program

$$(x := x + 1; y := y + 1) \mathbin{\vert\vert\vert} (y := y + 2; x := x + 2).$$

In practice, if the two programs run on different processors, there will be data race between them, when we intend to interpret such program as a false formula and indicate a programming fault. But with the new parallel operator $\vert\vert\vert$, since there is no atomic blocks, the program is equivalent to

$$(\bigcirc x = x + 1 \wedge \bigcirc(\bigcirc y = y + 1)) \wedge (\bigcirc y = y + 2 \wedge \bigcirc(\bigcirc x = x + 2)),$$

which can still evaluate to true.

3.2 An Axiom System for MSVL

In [27], the axiomatic system of MSVL is formalized in two parts : one for state deduction and the other for interval deduction. In state deduction, state axioms and state inference rules are given. In interval deduction, a variation of Hoare's triple that is $\{\sigma_k, A\}\ p\ \{\sigma_h, B\}$ is proposed as correctness assertion. In this section, we add some new axioms and inference rules to specify the pointer assignment, atomic blocks and parallel statements, and prove that the axiomatic system is sound as well. With this axiomatic, a non-blocking system is modeled using MSVL language. Then, properties of the system are specified by the interval-based assertion language defined in Sect. 1. Hence, the model and property of a system can be written in one logic $\alpha\mathsf{PTL}$.

Axioms and Inference Rules Within a State. We first give the state axioms and state inference rule for the extended MSVL. Particularly, state axioms $(A9), (A12)$, $(A17 - A22)$ and $(Atom)$ are new ones for the parallel statement and atomic blocks. For convenience, we abbreviate $\vdash \Box(p \leftrightarrow q)$ as $p \cong q$.

State formulas ps and extended state formulas Qs are defined as follows:

$$\begin{aligned} ps\ &::= x = e \mid x \Leftarrow e \mid ps \wedge ps \mid \mathsf{lbf}(x) \mid \mathsf{True} \\ Qs &::= ps \mid \langle p \rangle \mid Qs \wedge Qs \end{aligned}$$

Definition 1 (Semi-Normal Form). *An MSVL program is* semi-normal form *if it has the following form*

$$q \ \cong \ (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$$

where Qs_{ci} and Qs_{ej} are extended state formulas for all i, j, p_{fi} is an MSVL program, $n_1 + n_2 \geq 1$.

Definition 2 (Normal Form). *The normal form of program q is defined as*

$$q \ \cong \ \bigvee_{i=1}^{k} (q_{ei} \wedge \varepsilon) \vee \bigvee_{j=1}^{h} (q_{cj} \wedge \bigcirc q_{fj})$$

where $k + h \geq 1$, q_{fj} is a general program in MSVL while q_{ei} and q_{cj} are true or all state formulas of the form : $(x_1 = e_1) \wedge \ldots \wedge (x_l = e_1) \wedge \dot{p}_{x_1} \wedge \ldots \wedge \dot{p}_{x_l}$, where $e_k \in D$ $(1 \leq k \leq D)$, $\dot{p}_x$ denotes p_x or $\neg p_x$.

Normal Form. plays an important role for deducing programs in MSVL. To deduce a program to its normal form, a set of state axioms and state inference rules is given in Tables 1, 2 and 3. In Tables 2, ps_1 and ps_2 are state formulas, $Qs_1 \equiv ps_1 \wedge \bigwedge_{i=1}^{l_1} \langle q_i \rangle$ and $Qs_2 \equiv ps_2 \wedge \bigwedge_{i=1}^{l_2} \langle q_i' \rangle$ are extended state formulas $(l_1, l_2 \geq 0)$, T_i denotes $\bigcirc p_i$ or ε $(i = 1, 2)$.

Lemma 2. *Semi-normal form can be deduced into its normal form by the state axioms.*

Table 1. State axioms.

$(A1)$	$\mathsf{lbf}(x) \wedge x = e \cong x = e \wedge p_x$ (where $\ominus x \neq e$)
$(A2)$	$\mathsf{lbf}(x) \wedge x \Leftarrow e \cong x = e \wedge p_x$
$(A3)$	$\Box p \wedge \mathsf{more} \cong p \wedge \bigcirc\Box p$
$(A4)$	$\Box p \wedge \varepsilon \cong p \wedge \varepsilon$
$(A5)$	$\mathsf{frame}(x) \wedge \mathsf{more} \cong \bigcirc(\mathsf{lbf}(x) \wedge \mathsf{frame}(x))$
$(A6)$	$\mathsf{frame}(x) \wedge \varepsilon \cong \varepsilon$
$(A7)$	$\bigcirc p \ ; q \cong \bigcirc(p \ ; q)$
$(A8)$	$\varepsilon \ ; q \cong q$
$(A9)$	$(Qs \wedge p) \ ; q \cong Qs \wedge (p \ ; q)$
$(A10)$	$(p_1 \vee p_2) \ ; q \cong (p_1 \ ; q) \vee (p_2 \ ; q)$
$(A11)$	$\bigcirc p \wedge q \cong \bigcirc p \wedge q \wedge \mathsf{more}$
$(A12)$	$\langle p_1 \vee p_2 \rangle \cong \langle p_1 \rangle \vee \langle p_2 \rangle$
$(A13)$	$x := e \cong \bigcirc(x = e \wedge \varepsilon)$
$(A14)$	$\mathtt{if}\ b\ \mathtt{then}\ p\ \mathtt{else}\ q \cong (b \wedge p) \vee (\neg b \wedge q)$
$(A15)$	$\mathtt{while}\ b\ \mathtt{do}\ p \cong \mathtt{if}\ b\ \mathtt{then}\ (p \wedge \mathsf{more}\ ; \mathtt{while}\ b\ \mathtt{do}\ p)\ \mathtt{else}\ \varepsilon$
$(A16)$	$\vdash P$, where P is a substitution instance of all valid formulas.

Proof. Let $Q_s \wedge \bigcirc q$ be a semi-normal form, where $Q_s \overset{\text{def}}{=} ps \wedge \bigwedge_{i=1}^{l} \langle q_i \rangle$. The proof can be done in the following two cases.

1. If $l = 0$, i.e., Q_s does not contain atomic blocks, then the semi-normal form $Q_s \wedge \bigcirc q$ is in normal form.
2. If $l \neq 0$, then by state axioms, we have

$$\begin{aligned} & Q_s \wedge \bigcirc q \\ \cong\ & ps \wedge \bigwedge_{i=1}^{l} \langle q_i \rangle \wedge \bigcirc q \\ \cong\ & ps \wedge (ps' \wedge \bigcirc p') \wedge \bigcirc q \quad (Axiom - Atm) \end{aligned}$$

□

Theorem 3. *Any MSVL program with atomic blocks can be deduced to its normal form by the state axioms and state inference rules.*

Proof. The proof proceeds by induction on the structure of the statements in MSVL with atomic blocks. Here we only prove the new extended statements that are parallel statement $p \mid\mid\mid q$, atomic block $\langle p \rangle$, pointer assignment $*Z_v = e$ and unit assignment $x := e$. The proofs of other statements are the same as in [27].

1. If the program is the parallel statement $p \mid\mid\mid q$, by axioms (A17-A22), we have $p \mid\mid\mid q$ can be deduced into its semi-normal form. Further, by Lemma 2, any semi-normal form can be deduced into its normal form. Thus, we know that $p \mid\mid\mid q$ has its normal form.

2. If the program is the atomic block $\langle p \rangle$, by axiom $(Axiom - Atm)$, we can deduce $\langle p \rangle$ into its normal form.
3. If the program is the pointer assignment $*Z_v = e$, we have

$$\begin{aligned} *Z_v = e &\cong *Z_v = e \land \mathsf{True} && \text{(A16,R3)} \\ &\cong *Z_v = e \land (\varepsilon \lor \bigcirc \mathsf{True}) && \text{(A16,R3)} \\ &\cong (*Z_v = e \land \varepsilon) \lor (*Z_v = e \land \bigcirc \mathsf{True}) && \text{(A16,R3)} \end{aligned}$$

4. If the program is the unit assignment $x := e$, by axiom (A13), we have $x := e \cong \bigcirc(x = e \land \varepsilon)$

□

Table 2. State axioms for the parallel statement and atomic blocks.

$(A17)$	$(Qs_1 \land \bigcirc p_1) \vert\vert\vert (Qs_2 \land \bigcirc p_2) \cong (Qs_1 \land Qs_2) \land \bigcirc (p_1 \vert\vert\vert p_2)$, if $V_{q_i} \cap V_{q'_i} = \varnothing$
$(A18)$	$(Qs_1 \land \bigcirc p_1) \vert\vert\vert (Qs_2 \land \bigcirc p_2) \cong (Qs_1 \land ps_2) \land \bigcirc (p_1 \vert\vert\vert (\bigwedge_{i=1}^{l_2} \langle q'_i \rangle \land \bigcirc p_2))$ $\lor (Qs_2 \land ps_1) \land \bigcirc (p_2 \vert\vert\vert (\bigwedge_{i=1}^{l_1} \langle q_i \rangle \land \bigcirc p_1))$, if $V_{q_i} \cap V_{q'_i} \neq \varnothing$
$(A19)$	$(Qs_1 \land \varepsilon) \vert\vert\vert (Qs_2 \land T_2) \cong (Qs_1 \land Qs_2) \land T_2$, if $(l_1 = l_2 = 0)$ or $(l_1 = 0$ and $l_2 > 0$ and $T_2 = \bigcirc p_2)$
$(A20)$	$(Qs_1 \land \varepsilon) \vert\vert\vert (Qs_2 \land T_2) \cong Qs_1 \land \varepsilon$, if $(l_2 > 0$ and $T_2 = \varepsilon)$
$(A21)$	$(Qs_1 \land \varepsilon) \vert\vert\vert (Qs_2 \land T_2) \cong Qs_2 \land T_2$, if $(l_1 \neq 0)$
$(A22)$	$(Qs_1 \land T_1) \vert\vert\vert (Qs_2 \land \varepsilon) \cong (Qs_2 \land \varepsilon) \vert\vert\vert (Qs_1 \land T_1)$
$(Atom)$	$\langle p \rangle \cong ps \land \bigcirc (ps_n \vert_{I_{prop}})$, where $ps, ps_n \overset{\text{def}}{=} \bigwedge_{i=1}^{n} ((x_i = e_i \land \dot{p}_{x_i})$ if $p \cong ps \land \bigcirc p_1$, $p_1 \cong ps_1 \land \bigcirc p_2, \cdots$ $p_{n-1} \cong ps_{n-1} \land \bigcirc p_n$, $p_n \cong ps_n \land \varepsilon$

Table 3. State inference rules.

$(R1)$	$p \cong q \Longrightarrow prog[p] \cong prog[q/p]$
$(R2)$	$p(x) \cong (p_e(x) \land \varepsilon) \lor (p_c(x) \land \bigcirc p_f(x))$ $\Longrightarrow \exists x : p(x) \cong (\exists x : p_e(x) \land \varepsilon) \lor (\exists x : p_c(x) \land \bigcirc \exists x : p_f(x))$
$(R3)$	$\vdash P \Longrightarrow \ \vdash \Box P$, where P is a substitution instance of all valid formulas.

Axioms and Inference Rules Over Intervals. In order to deduce programs over intervals, the correctness assertion for specifying and verifying properties is given in [27]. In the paper, we formalize $(Rule - Atom)$, $(Rule - PointerI)$, and $(Rule - PointerII)$ for the atomic blocks and the pointer assignment.

Definition 3. The correctness assertion is defined as follows.

$$\{\sigma_k, A\} \ p \ \{\sigma_h, B\}$$

where p is an MSVL program, σ_k and σ_h are intervals, and σ_k is a prefix of σ_h $(k, h \in N_\omega, 0 \leq k \leq h)$, A and B are assertions that can be used in the following manner:

(1) For partial correctness properties, A and B are required to be *state formulas and state assertions.*
(2) For run-time properties such as *eventually* ($\Diamond$), *always* ($\Box$), A is a *temporal formula*, and B *a state formula.*
(3) If p is non-terminating, B should be True.

Particularly, when $k = 0$, $\{\sigma_0, A\}\ p\ \{\sigma_h, B\}$ is an *initial triple*, in which program p is deduced over interval σ_h starting from initial state s_0. For convenience, in the axiomatic system, we abbreviate $\{\sigma_0, A\}\ p\ \{\sigma_h, B\}$ as $\{A\}\ p\ \{B\}$ and use (σ_h, k) to denote $\sigma_{(k\ldots h)}$.

Table 4. Selected interval inference rules.

(ISR)	$\{\sigma_k, A\}prog[p]\{\sigma_h, B\}$ and $p \cong q$ $\Longleftrightarrow \{\sigma_k, A\}prog[q/p]\{\sigma_h, B\}$ and $p \cong q$
(AAS)	$\{\sigma_k, A\}\ x = m \wedge ps(y) \wedge p\ \{\sigma_h, B\}$ $\Longleftrightarrow \{\sigma_k, A\}\ x = m \wedge ps(y)[x \mapsto m] \wedge p\ \{\sigma_h, B\}$ where $\mathsf{lbf}(y)$ and $\mathsf{lbf}(x)$ does not occur in p.
(LBF)	$\{\sigma_k, A\}\ \mathsf{lbf}(x) \wedge p\ \{\sigma_h, B\}$ $\Longleftrightarrow \{\sigma_k, A\}\ x = \sigma_{k-1}(x) \wedge p\ \{\sigma_h, B\}$ where $x = e$ and $x \Leftarrow e$ does not occur in p.
(ANext)	$\{\sigma_k, A\}\ p_c \wedge \bigcirc p_f\ \{\sigma_h, B\}$ $\Longleftrightarrow \begin{cases} \{\sigma_k[p_c] \cdot \langle s_{k+1}\rangle, A_f\}\ p_f\ \{\sigma_h, B\} \\ \text{ and } \{\sigma_k, A_c\}\ p_c\ \{\sigma_k, A_c\},\ \text{if } p_c \to A_c \\ \{\sigma_k, \mathsf{False}\}\ p_c \wedge \bigcirc p_f\ \{\sigma_h, B\} \\ \text{otherwise} \end{cases}$ where $A \equiv (A_c \wedge \bigcirc A_f) \vee (A_e \wedge \varepsilon)$

Definition 4. *Let Σ be a set of all intervals, and $(\sigma_h, k) = \langle \delta_k, \ldots, \delta_h \rangle$ be a subinterval of σ_h starting from state δ_k $(0 \leq k \leq h)$, where $\sigma_h = \sigma_k \circ (\sigma_h, k)$. We say that:*

- $\{\sigma_k, A\}\ p\ \{\sigma_h, B\}$ *is* satisfied *over σ_h starting from δ_k, denoted by*

$$(\sigma_h, k) \models \{\sigma_k, A\}\ p\ \{\sigma_h, B\}$$

  if $(\sigma_h, 0, k, h) \models p$, *then* $(\sigma_h, 0, k, h) \models A$ *and* $(\sigma_h, 0, h, h) \models B$.
- *If for all $\sigma_h \in \Sigma$, $(\sigma_h, k) \models \{\sigma_k, A\}\ p\ \{\sigma_h, B\}$, then $\{\sigma_k, A\}\ p\ \{\sigma_h, B\}$ is* valid*, denoted by $\models \{\sigma_k, A\}\ p\ \{\sigma_h, B\}$.*
- *$\{A\}\ p\ \{B\}$ is* satisfiable*, denoted by $(\sigma_h, 0) \models \{A\}\ p\ \{B\}$, if $(\sigma_h, 0) \models \{\sigma_0, A\}\ p\ \{\sigma_h, B\}$.*
- *If for all $\sigma_h \in \Sigma$, $(\sigma_h, 0) \models \{A\}\ p\ \{B\}$, then $\{A\}\ p\ \{B\}$ is* valid*, denoted by $\models \{A\}\ p\ \{B\}$.*

Table 5. Interval rule for the atomic blocks and the pointer assignment.

(*Rule-Atom*)	$\{\sigma_k, A\}\ x = m \wedge \langle Q\rangle \wedge ps(y) \wedge p\ \{\sigma_h, B\}$ $\Longleftrightarrow \{\sigma_k, A\}\ x = m \wedge \langle Q[x \mapsto m]\rangle \wedge ps(y)[x \mapsto m] \wedge p\ \{\sigma_h, B\}$
(*Rule-PointerI*)	$\{\sigma_k, A\}\ *Z_v = e\ \{\sigma_h, B\}$ $\Longleftrightarrow \begin{cases} \{\sigma_k, A\}\ \gamma(Z) = e\ \{\sigma_h, B\} & \text{if } Z_v = Z \in \mathsf{PVar} \\ \{\sigma_k, A\}\ y = e\ \{\sigma_h, B\} & \text{if } Z_v = \&y \in \mathsf{PConst} \end{cases}$
(*Rule-PointerII*)	$\{\sigma_k, A\}\ x = Z_v\ \{\sigma_h, B\}$ $\Longleftrightarrow \begin{cases} \{\sigma_k, A\}\ x = \gamma(Z)\ \{\sigma_h, B\} & \text{if } Z_v = Z \in \mathsf{PVar} \\ \{\sigma_k, A\}\ x = y\ \{\sigma_h, B\} & \text{if } Z_v = \&y \in \mathsf{PConst} \end{cases}$

Table 4 regards to some of selected interval inference rules, where rule (ISR) is a congruence rule; rules (AAS) and (LBF) focus on evaluating arithmetic expressions; rule (ANext) is about the progress of normal form.

Table 5 presents a series of interval inference rules for atomic blocks and pointer operations. $(Rule - Atom)$ evaluates the variables inside the atomic blocks; $(Rule - PointerI)$ and $(Rule - PointerII)$ are used to evaluate the pointer assignment. There are two cases need to be discussed: one is for $Z_v \in \mathsf{PVar}$ and the other is for $Z_v \in \mathsf{PConst}$. Both of them can be deduced according to their definitions.

Definition 5. *Some notations are given as follows.*

$$(1)\quad ps(y)[x \mapsto m] \overset{\text{def}}{=} \begin{cases} b(x)[x \mapsto m] & \textit{if } ps(y) \textit{ is } b(x) \\ y = e(x)[x \mapsto m] & \textit{if } ps(y) \textit{ is } y = e(x) \\ y = e(x)[x \mapsto m] \wedge p_y & \textit{if } ps(y) \textit{ is } y \Leftarrow e(x) \\ ps(y) & \textit{if } x \textit{ does not occur in } ps(y). \end{cases}$$

$$(2)\quad \langle Q\rangle[x \mapsto m] \overset{\text{def}}{=} \langle ps(y)[x \mapsto m] \wedge p\rangle, if Q \cong ps(y) \wedge p$$

$$(3)\quad \bigwedge_{i=1}^{n}((x_i = e_i \wedge \dot{p}_{x_i})\ |_{I_{prop}}) \overset{\text{def}}{=} \bigwedge_{i=1}^{n}(x_i = e_i)$$

Theorem 4. *The axiom system is soundness.*

Let A be an axiom or an inference rule in the axiomatic systems. We need to prove $\vdash A \rightarrow\ \models A$. The proof can proceed by proving each axiom and inference rule valid in the model semantics. The details of the proof is omitted here.

4 Verification of Treiber's Lock-Free Stack

Treiber's stack is a non-blocking stack including two operations *pop*() and *push*(x), which is implemented as a linked list pointed by the *Top* pointer. It allows simultaneous read and write of *Top*, and the conflict detection is done by the CAS (compare-and-swap) command. The codes of Treiber's stack are presented in Table 6.

Fine-grained concurrency algorithms are usually implemented with basic machine primitives, whose atomicity is enforced by the hardware. For instance, the machine primitive $\mathsf{CAS}(\mathsf{x},\ \mathsf{old},\ \mathsf{new};\ \mathsf{ret})$ can be defined in $\alpha\mathsf{PTL}$ as follows, which has the meaning : if the value of the shared variable x is equivalent to old then to update it with new and return 1, otherwise keep x unchanged and return 0.

Table 6. Treiber's stack.

pop(){
local done, next, t;
done := *false*;
`while`(!*done*) {
  t := *Top*;
  `if`(*t* := *null*) *return null*;
  next := *t.next*;
  done := CAS(*Top*, *t*, *next*);
  }
return t;
}
	push(*x*){
	local done, t;
	done := *false*;
	`while`(!*done*){
	  t := *Top*;
	  x.next := *t*;
	  done := CAS(*Top*, *t*, *x*);
	  }
	return true;
	}

In real systems, the atomicity of CAS is enforced by the hardware. Note that, in our logic, we use the atomic interval formula $(\langle p\rangle)$ to assure the atomicity. CAS primitive can be defined in MSVL as follows:

$$\begin{aligned}\mathsf{CAS}(x,\mathsf{old},\mathsf{new};\mathsf{ret}) &\overset{\text{def}}{=} \langle\ \mathtt{if}\ (x=\mathsf{old})\ \mathtt{then}\ x:=\mathsf{new}\wedge\mathsf{ret}:=1)\ \mathtt{else}\ \mathsf{ret}:=0\ \rangle\\ &\equiv \langle(x=\mathsf{old}\ \wedge x:=\mathsf{new}\wedge\mathsf{ret}\ :=1)\vee(\neg(x=\mathsf{old}\ )\wedge\mathsf{ret}:=0)\rangle\\ &\equiv \langle(x=\mathsf{old}\ \wedge\bigcirc(x=\mathsf{new}\wedge\mathsf{ret}\ =1\wedge\varepsilon))\vee(\neg(x=\mathsf{old}\ )\wedge\bigcirc(\mathsf{ret}\ =0\wedge\varepsilon))\rangle\end{aligned}$$

Each node in the linked list can be presented as a *list*, like $node_i = \langle item, \&node_{i+1}\rangle$. For example, $node_1 = \langle 6, \&node_2\rangle$. Null pointer is described by -1. We use symbol [ ] to get the elements of nodes. That is, $node_i[0] = item$ and $node_i[1] = \&node_{i+1}$.

Therefore, we can now specify the Treiber's stack using MSVL, which is shown in Fig. 7. Operations *pop* and $push(x)$ in Table 6 are described by the pop_i and $push_i(arg)$ respectively, where arg is a parameter of the $push$ operation and i denotes the thread id. Both of pop_i and $push_i(arg)$ are implemented based on the non-blocking synchronization. Treiber's stack has a lock-free property that means some method call finishes in a finite number of steps. We will take $Prog$ in Fig. 7 as an example to illustrate when two operations $push\ |||\ push$ implement concurrently, the $Prog$ satisfies the lock-free property.

The lock-free property can be specified in our logic as $\Box\Diamond(done_1 = 1 \vee done_2 = 1)$, which means that some thread can finish the operation $push(x)$ in finite steps. Let $A = \Box\Diamond(done_1 = 1 \vee done_2 = 1)$ and $B = \mathsf{True}$. In the following we will prove that $Prog$ satisfies A in our axiomatic system $\{\sigma_0, A\}\ Prog\ \{\sigma, B\}$.

The outline of the proof proceed by induction on the number of states. However, in the paper, we will focus mainly on how to use the axioms and inference

```
$pop_i \overset{\text{def}}{=}$ frame$(t_i, done_i, next_i, ret_i) \land done_i =$ False $\land$ while $(!done_i)$ do
        {
          $t_i = Top \land$ if $(t_i = -1)$ then $ret_i := -1$;
          $next_i = (*t)[1] \land$ CAS$(Top, t_i, next_i; done_i) \land$ skip;
          if $(done_i = 1)$ then $ret_i := (*t)[0]$
        }

$push_i(arg) \overset{\text{def}}{=}$ frame$(t_i, done_i, arg[1], ret_i) \land done_i =$ False $\land$ while $(!done_i)$ do
        {
          $t_i = Top \land arg[1] = t_i \land$ CAS$(Top, t_i, \&arg; done_i) \land$ skip;
          if $(done_i = 1)$ then $ret_i := 1$ else $ret_i := 0$
        }

$Prog \overset{\text{def}}{=}$ frame$(Top) \land Top = -1 \land$ while (True) do $\{push_1(x) \ ||| \ push_2(y)\}$
```

Fig. 7. Treiber's stack in MSVL.

rules of atomic blocks, the parallel statement and the pointer assignment to reason about programs.

(1) Firstly, by the state axioms $(A1-A16)$ and state inference rules $(R1-R3)$, we deduce the $push_i(arg)$ into its semi-normal form as follows.

$$\begin{aligned} push_i(arg) \cong\ & (done_i = 0) \land (t_i = \mathit{Top} \land arg[1] = t_i) \land \\ & \mathsf{CAS}(\mathit{Top}, t_i, \&arg; done_i) \land \bigcirc push_i'(arg) \end{aligned}$$

where

$$\begin{aligned} push_i'(arg) \cong\ & \mathsf{frame}(t_i, done_i, arg[1], ret_i) \land \mathsf{lbf}(t_i, done_i, arg[1], ret_i) \land \\ & ((done_i = 1 \land ret_i := 1 \lor done_i = 0 \land ret_i := 0) \ ; \texttt{while} \ (\neg done_i) \ \texttt{do} \ P_1 \end{aligned}$$

(2) Secondly, let arg be x and y respectively. Using state axioms $(A17-A22)$, we deduce $push_1(x) \ ||| \ push_2(y)$ into its semi-normal forms as follows.

$$\begin{aligned} push_1(x) \ ||| \ push_2(y) \cong\ & (done_1 = 0) \land (done_2 = 0) \land (t_1 = \mathit{Top} \land x[1] = t_1) \land \\ & (t_2 = \mathit{Top} \land y[1] = t_2) \land \mathsf{CAS}(\mathit{Top}, t_1, \&x; done_1) \land \\ & \bigcirc(push_1'(x) \ ||| \ \mathsf{CAS}(\mathit{Top}, t_2, \&y; done_2) \land \bigcirc push_2'(y)) \\ & \lor \\ & (done_1 = 0) \land (done_2 = 0) \land (t_1 = \mathit{Top} \land x[1] = t_1) \land \\ & (t_2 = \mathit{Top} \land y[1] = t_2) \land \mathsf{CAS}(\mathit{Top}, t_1, \&x; done_1) \land \\ & \bigcirc(push_2'(y) \ ||| \ \mathsf{CAS}(\mathit{Top}, t_1, \&x; done_1) \land \bigcirc push_1'(x)) \end{aligned}$$

(3) Finally, let us go back to the proof framework $\{\sigma_0, A\}\ Prog\ \{\sigma_h, B\}$. By Lemma 2 and Theorem 3, we can deduce $Prog$ into its semi-normal form and normal form, which are denoted by $SNF(Prog)$ and $NF(Prog)$ respectively. That is, we have,

$$\begin{aligned} & \{\sigma_0, A\}\ Prog\ \{\sigma_h, B\} \\ \cong\ & \{\sigma_0, A\}\ SNF(Prog)\ \{\sigma_h, B\} \\ \cong\ & \{\sigma_0, A\}\ NF(Prog)\ \{\sigma_h, B\} \end{aligned}$$

where $SNF(Prog)$ can be deduced from $Prog$ by state axioms and state inference rules. Some of deduction steps are shown as follows.

$$
\begin{aligned}
\mathit{Prog} \overset{\text{def}}{=}\;& \mathsf{frame}(\mathit{Top}) \wedge \mathit{Top} = -1 \wedge \mathtt{while}(\mathsf{True})\ \mathtt{do}\ \{\mathit{push}_1(x) \wedge \mathit{push}_2(y)\} \\
\cong\;& \mathsf{frame}(\mathit{Top}) \wedge \mathit{Top} = -1 \wedge (\mathit{push}_1(x) \mid\mid\mid \mathit{push}_2(y) \wedge \mathsf{more}\ ; \\
& \mathtt{while}\ (\mathsf{True})\ \mathtt{do}\ \{\mathit{push}_1(x) \wedge \mathit{push}_2(y)\}) \\
& \vdots \\
\cong\;& (\mathit{Top} = -1) \wedge (\mathit{done}_1 = 0) \wedge (\mathit{done}_2 = 0) \wedge (t_1 = \mathit{Top} \wedge x[1] = t_1) \wedge \\
& (t_2 = \mathit{Top} \wedge y[1] = t_2) \wedge \mathsf{CAS}(\mathit{Top}, t_1, \&x; \mathit{done}_1) \wedge \bigcirc \mathit{Prog}_1 \\
\vee\;& (\mathit{Top} = -1) \wedge (\mathit{done}_1 = 0) \wedge (\mathit{done}_2 = 0) \wedge (t_1 = \mathit{Top} \wedge x[1] = t_1) \wedge \\
& (t_2 = \mathit{Top} \wedge y[1] = t_2) \wedge \mathsf{CAS}(\mathit{Top}, t_2, \&y; \mathit{done}_2) \wedge \bigcirc \mathit{Prog}_2
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{Prog}_1 \overset{\text{def}}{=}\;& \bigcirc(\mathsf{frame}(\mathit{Top}) \wedge \mathsf{lbf}(\mathit{Top}) \wedge (\mathit{push}'_1(x) \mid\mid\mid \mathsf{CAS}(\mathit{Top}, t_2, \&y; \mathit{done}_2) \wedge \bigcirc \mathit{push}'_2(y))\ ; \\
& \quad \mathtt{while}\ (\mathsf{True})\ \mathtt{do}\ \{\mathit{push}_1(x) \wedge \mathit{push}_2(y)\}) \\
\mathit{Prog}_2 \overset{\text{def}}{=}\;& \bigcirc(\mathsf{frame}(\mathit{Top}) \wedge \mathsf{lbf}(\mathit{Top}) \wedge (\mathit{push}'_2(y) \mid\mid\mid \mathsf{CAS}(\mathit{Top}, t_1, \&x; \mathit{done}_1) \wedge \bigcirc \mathit{push}'_1(x))\ ; \\
& \quad \mathtt{while}\ (\mathsf{True})\ \mathtt{do}\ \{\mathit{push}_1(x) \wedge \mathit{push}_2(y)\})
\end{aligned}
$$

Thus, we obtain the semi-normal form of Prog. Furthermore, by axiom (Atom), we can unfold CAS into the form like $ps \wedge \bigcirc ps'$, which results in the normal form of Prog. By rule (ANext), we can rewrite the program from the current state to the next one and meanwhile verify whether the current state satisfies A. The process can be done in the semi-automatical verification tool PVS [27]. We do not present the details in the paper due to the limited space.

5 Conclusion

This paper proposes an interval-based assertion language that combines separation logic and projection temporal logic to reason about the non-blocking programs with shared mutable data structures. We employ MSVL as the model language and the interval-based assertion language as the specification language such that the algorithms and properties can be specified in one notation. To well write the non-blocking algorithms in MSVL, we also investigate a new concurrent temporal logic model that extends MSVL with the atomic blocks, the parallel statement and the pointer assignment. Furthermore, we augment our axiomatic system with the axioms and inference rules for these new structures in MSVL. In addition, we use Treiber's stack as an example to illustrate both the MSVL and the axiomatic system can be well used. However, in the paper, we do not particularly investigate the correctness of the pointer programs. In the future, we will focus on this problem and verify the issues such as the alias name in pointer programs by means of our assertion language. In addition, the completed proof of Theorem 4 and the Treiber's stack will be found in our technical report.

References

1. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier, Amsterdam (2009)
2. Ashcroft, Edward, A.: Proving assertions about parallel programs. J. Comput. Syst. Sci. **10**(1), 110–135 (1975)
3. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: PLDI, pp. 338–349. ACM Press, New York (2003)

4. Cohen, E., Lamport, L.: Reduction in TLA. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 317–331. Springer, Heidelberg (1998)
5. Jacobs, B., Piessens, F., et al.: Safe concurrency for aggregate objects with invariants. In: Proceedings of the 3rd IEEE Conference on Software Engineering and Formal Methods. pp. 137–147. (2005)
6. Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A statically verifiable programming model for concurrent object-oriented programs. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 420–439. Springer, Heidelberg (2006)
7. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. Electron. Notes Theor. Comput. Sci. **137**(2), 93–110 (2005)
8. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
9. Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 173–188. Springer, Heidelberg (2007)
10. Feng, X.: Local rely-guarantee reasoning. In: POPL, pp. 315–327. ACM Press, New York (2009)
11. Vafeiadis, V.: Modular Fine-Grained Concurrency Verification. Cambridge University, Cambridge (2008)
12. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
13. Pnueli, A.: The temporal semantics of concurrent programs. In: Proceedings of the International Symposium Semantics of Concurrent Computation. LNCS, vol. 70, pp. 1–20. Springer-Verlag (1979)
14. Abadi, M., Manna, Z.: Temporal logic programming. J. Symbolic Comput. **8**(1–3), 277–295 (1989)
15. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
16. Rondogiannis, P., Gergatsoulis, M., Panayiotopoulos, T.: Branching-time logic programming: the language cactus and its applications. Comput. Lang. **24**(3), 155–178 (1998)
17. Moszkowski, B.C.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)
18. Duan, Z., Yang, X., Koutny, M.: Semantics of framed temporal logic programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 356–370. Springer, Heidelberg (2005)
19. Yang, X., Duan, Z.: Operational semantics of framed tempura. J. Log. Algebraic Program. **78**(1), 22–51 (2008)
20. Duan, Z., Koutny, M.: A framed temporal logic programming language. J. Comput. Sci. Technol. **19**(1), 341–351 (2004)
21. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Informatic **45**, 43–78 (2008)
22. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society (2002)
23. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). and 583
24. OHearn, P.W.: Resources, concurrency and local reasoning. Theor. Comput. Sci. **375**(1–3), 271–307 (2007)

25. Yang, X., Zhang, Y., Fu, M., Feng, X.: A concurrent temporal programming model with atomic blocks. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 22–37. Springer, Heidelberg (2012)
26. Wang, X., Duan, Z.: Pointers in framing projection temporal logic programming language. J. Xidian Univ. **36**(5), 1069–1074 (2008)
27. Yang, X., Duan, Z., Ma, Q.: Axiomatic semantics of projection temporal logic programs. Math. Struct. Comput. Sci. **20**(5), 865–914 (2010)

Semantics and Analysis

Formal Semantics of Orc Based on TLA$^+$

Zhen You[1,2](✉), Jinyun Xue[1,2], Qimin Hu[1,2], and Yi Hong[3]

[1] Provincial Key Laboratory for High-Performance Computing Technology,
Jiangxi Normal University, Nanchang, China
{youzhenjxnu,qiminhu}@163.com, jinyun@vip.sina.com

[2] State International S&T Cooperation Base of Networked Supporting Software,
Jiangxi Normal University, Nanchang, China

[3] Department of Computer Science, University of Leicester, Leicester, UK
yh37@mcs.le.ac.uk

Abstract. Concurrency is ubiquitous today. Orc provides four powerful combinators (parallel combinator, sequential combinator, pruning combinator and otherwise combinator), used to structured concurrent programming in a simple and hierarchical manner. In order to extend concurrent mechanism in our abstract sequential programming language, called *Apla*, we have already done some research about Orc. The paper takes a step towards this goal by presenting formal semantics of Orc based on TLA$^+$ language. Compared with other semantics of Orc, our major concern is Orc expression's next-state relation/action, which is ideal for expressing behavior of a sequence of states. And liveness properties of Orc expression are also elaborated by using TLA$^+$ weak fairness. After analysis and comparison, our proposal could be simpler to illustrate specification of Orc program via the well known dining philosophers problem.

Keywords: Concurrency · Orc combinator · Apla · Formal semantics · TLA$^+$

1 Introduction

Concurrency has become an essential characteristics of programming with the advent of multi-core system and the widespread applications of Internet. Unfortunately, concurrency contributes mightily to complexity in programming [1], and it is difficult for programmer. Compared with some typically abstract concurrent language (such as RSL, Circus, and SCOOP+Effiel), we discovered that Orc theory [1–4] and its practical applications [5] could express orchestrations and wide-area computations in a simple and constructed manner. The major

This work was supported in part by the National Nature Science Foundation of China (Grant No.61462041,61472167,61462039,61272075), the major international cooperative research project (Grant No.61020106009) from National Natural Science Foundation of China, the National Natural Science Foundation of Jianxi Province (Grant No.20132BAB211022,20142BAB217023) and the Science and Technology Research Project of Jianxi Province Educational Department(Grant No.GJJ13231,GJJ14268).

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 147–163, 2015.
DOI: 10.1007/978-3-319-17404-4_10

aspect of Orc is on its insistence on concurrency in orchestration of components, including web services. Its initial goal is to use sequential components at the lowest-level, and orchestrate them, possibly, concurrently [1].

Orc was originally presented as a process calculus, it has now evolved into *Orc theory*, consists of Orc calculus and Orc programming language. Besides of simplicity, Orc has powerful expression ability, because it could solve all known synchronization and communication problems, and solve all known forms of real-time and periodic computations [6]. Orc provides four powerful combinators (parallel combinator, sequential combinator, pruning combinator and otherwise combinator) that define the structure of a concurrent computation. These combinators, which are inherently concurrent, support sequential and concurrent execution, concurrent execution with blocking and termination.

Logic is particularly important because it is the mathematics basis of software: it is used to formalized the semantics of programming language and the specification of programs and to verify the correctness of programs [7]. *Temporal logic* with feature of reasoning and specifying behaviors of both hardware and software components, is firstly introduced by Amir Pnueli in 1977 [8]. In order to describing more properties of systems, Leslie Lamport invented TLA [9,10], the Temporal Logic of Actions – a simple variant of Pnueli's original logic. And then he refined TLA$^+$ [11] language to writing specifications of program interfaces (APIs), distributed systems and asynchronous systems. Besides providing a mathematical foundation for describing systems, TLA$^+$ also provides a nice way to formalize the style of reasoning about systems [12] that has proved to be most effective in practice – a style known as *assertional reasoning*.

Based on our original research results – abstraction sequential programming language *Apla* [13] from *PAR method and PAR platform* [14–16], our main research work at present is to implement Orc concurrent mechanism in *Apla* Language, so that it becomes a sequential and concurrent language, called *Apla*$^+$. In order to achieve this goal, we have done some research on Orc theory and its concurrent combinators. In this paper, we proposed formal semantics of Orc's four combinators based on TLA$^+$ language, and the semantics would be easier to be modeled, simulated and verified using MSVL language [17–19] and its toolkit MSV. Difference from other Orc's operational or denotational semantics [20–24], our major contribution is Orc expression's next-state relation/action and its liveness properties by using TLA$^+$ language.

Why TLA$^+$ notations were selected for describing semantics of Orc combinators? TLA$^+$ is a logic that combines temporal logic and logic of actions, which can be used to specify and reason about concurrent features of Orc expressions (combined by Orc combinators), and specify their safety and liveness properties. On the other hand, TLA$^+$ has structural concepts, such as modular decomposition and refinement of specifications. Thirdly, there are some useful supporting tools, including model-checker TLC, prover TLAPS.

The paper is structured as follows: Sect. 2 reviews some foundational knowledge about Orc theory and TLA$^+$ language. Our original research work is elaborated in Sect. 3, where each sub-section presents TLA$^+$ semantics of Orc combinator. Then we give a case study–classical dining philosophers problem in Sect. 4. Some related works are demonstrated and compared in Sect. 5. Finally, conclusion is discussed in last Section.

2 Background Knowledge

2.1 Orc Theory

A theory of programming, called Orc, developed by Professor Jayadev Misra and his collaborators. Orc introduced three important concepts, which are ***site***, ***expression*** and ***combinator***.

A ***site*** [1] is an even more general notion. It includes any program components that can be embedded in a larger program, some cyber-physical device and human being. A site is a service that is called like a traditional procedure and it responds with some number of values, called *publications*.

An Orc program is an expression. An ***expression*** [1] is either (1) a site call, (2) two constituent expressions combined using a *combinator*, or (3) a site definition followed by an expression. The expression following all the definitions in a program is called its goal expression. A program execution starts by executing its goal expression.

Orc has four combinators: parallel (|), sequential (>>), pruning (<<), and otherwise (;). A combinator forms an expression from two component expressions. *Each combinator captures a different aspect of concurrency.* Syntactically, the combinators are written infix, and have lower precedence than operators or calls, but higher precedence than other expression forms [25].

Combinator Precedence Level: sequential > parallel > pruning > otherwise

2.2 TLA$^+$ Language

Temporal logic has tow unary prefix basic operators: *always*, denoted $\Box$, and *eventually*, denoted $\Diamond$. The semantics of the logic is defined in terms of states, where a ***state*** is an assignment of values to program variables; and an infinite sequence of states is called a ***behavior***; it represents en execution of the program [12]. An ***action*** is a boolean-valued expression consisting of variables, primed variables, and constant symbols, and it defines a relation between on old and a new state.

TLA$^+$ specification is writhen to an abstract program, which consists of three things: (1) a set of possible initial states; (2) a next-state relation describing the states that may be reached in a single step from any given state; and (3) some fairness requirement [12].

$$\boldsymbol{Init} \wedge \Box[\boldsymbol{Next}]_{\mathbf{vars}} \wedge \boldsymbol{Liveness}$$

where ***Init*** is the initial predicate, used to describe all possible initial states. ***Next*** is the next-state action, used to describe the disjunction of all possible moves, and ***vars*** is the tuple of all variable. ***Liveness*** properties, described as temporal formulas, is the conjunction of formulas of the form weak fairness $\mathbf{WF}_{vars}(\boldsymbol{Next})$ and/or strong fairness $\mathbf{SF}_{vars}(\boldsymbol{Next})$, for Action ***Next***.

Some fundamental definitions and operators of TLA$^+$ [11] needed in this paper are given as follows.

▶ e' [The value of e in the final state of a step]
▶ $[A]_e$ $[A \vee (e' = e)]$
▶ $\langle A \rangle_e$ $[A \wedge (e' \neq e)]$
▶ **ENABLED** A [An A step is possible]
▶ **CHOOSE** $x : p$ [An x satisfying p]
▶ **CHOOSE** $x \in S : p$ [An x in S satisfying p]
▶ **CASE** $p_1 \rightarrow e_1 \blacksquare \ldots \blacksquare p_n \rightarrow e_n$ [Some e_i such that p_i true. In order to distinguish with *always*, we use ■ in **CASE** expression.]
▶ **CASE** $p_1 \rightarrow e_1 \blacksquare \ldots \blacksquare p_n \rightarrow e_n \blacksquare OTHER \rightarrow e$ [Some e_i such that p_i true, or e if all p_i are false]
▶ $\mathbf{WF}_e(A) \triangleq \Box(\Box\mathbf{ENABLED}\langle A \rangle_S \Rightarrow \Diamond\langle A \rangle_S)$ [A step must eventually occur if A is *continuously* enabled, *continuously* means without interruption.]
▶ $\mathbf{SF}_e(A) \triangleq \Box\Diamond\mathbf{ENABLED}\langle A \rangle_S \Rightarrow \Box\Diamond\langle A \rangle_S$ [A step must eventually occur if A is *continually* enabled, *continually* means repeatedly, possibly with interruptions.]
▶ **Module** ***Sequences***: ○ *Append Head Tail Len Seq SubSeq SelectSeq*

3 Formal Semantics of Orc Combinators

In order to explore concurrent-feature of four combinators, their formal semantics expressed using TLA$^+$ language are defined in this section.

3.1 Semantics of Parallel Combinator

Syntax of parallel combinator is described as:

$$Parallel ::= Expression | Expression$$

F | G means execution of expression F | G occurs by executing F and G immediately and concurrently. Each publication of F or G is published by F | G. Executions of F and G are interleaved arbitrarily in the execution of F | G. When both F and G have halted, F | G halts [25]. There is no direct communication between F and G during the execution [1].

Example 3.1. Expression *add(1,2)|sub(6,1)* calls the sites *add(1,2)* and *sub(6,1)* simultaneously. Since each of these sites publishes the corresponding value, expression *add(1,2)|sub(6,1)* publishes 3 and 5 immediately, in either order.

Supposing publications of F | G are stored in a sequence $\boldsymbol{S}$, we write an initial predicate $Init_1$ that specified the possible initial value of $\boldsymbol{S}$.

$$Init_1 \triangleq \boldsymbol{S} = \emptyset \tag{1}$$

A step of the execution F | G either sends a value of F or sends a value of G. The next-state relation $\mathcal{N}_1$ is defined that appending one value V_F or V_G at the end of sequence $\boldsymbol{S}$.

$$\mathcal{N}_1 \triangleq \boldsymbol{S}' = \boldsymbol{S} \circ \langle V_F \rangle \vee \boldsymbol{S}' = \boldsymbol{S} \circ \langle V_G \rangle \tag{2}$$

To express liveness $\mathcal{L}_1$, the specification is strengthened to the form, which means that a step must eventually occur if $\mathcal{N}_1$ is *continuously* enabled.

$$\mathcal{L}_1 \triangleq \mathbf{WF}_S(\mathcal{N}_1) = \Box(\Box ENABLED \;\; \langle \mathcal{N}_1 \rangle_S \Rightarrow \Diamond \langle \mathcal{N}_1 \rangle_S) \tag{3}$$

According to the formulas (1)–(3), it's easier to take as our formal specification of parallel expression F | G. Hence, its semantics ***Par*** defined by

$\boldsymbol{Par} : Init_1 \wedge \Box[\mathcal{N}_1]_S \wedge \mathcal{L}_1$ ***Semantics* 1**

3.2 Semantics of Sequential Combinator

Syntax of sequential combinator is described as:

$$Sequence ::= Expression > [Variable/Pattern] > Expression$$

There are three types of sequential expression:

$$\begin{aligned} &F > x > G && \text{[where x is a variable]} \\ &F > P > G && \text{[where P is a pattern]} \\ &F >> G && \end{aligned}$$

Those semantics are defined as following.

- **F >x> G**

In F >x> G, first the execution of F is started. Whenever F publishes a value, a new execution of G begins in parallel. The values published by F are consumed by the variable binding [1]. Each value published by F initiates a separate execution of G wherein x is bound to that published value. At the end, any value published by any executions of G is published by the whole expression F >x> G [25].

Example 3.2. In expression *add(1,2)|sub(6,1)* >x> *power(x)*, since sites *add(1,2)* and *sub(6,1)* execute simultaneously, it publishes 3 and 5 in either order. Site *power* is called twice with parameter x bound to 3 and 5, Therefore, the entire expression published 9 and 25 in either order.

Supposing publications of F >x> G are stored in a sequence $\boldsymbol{S}$, the initial predicate $Init_{2.1}$ is straightforward.

$$Init_{2.1} \triangleq \boldsymbol{S} = \emptyset \tag{4}$$

The next-relation $\mathcal{N}_{2.1}$ that specifies how the value of $\boldsymbol{S}$ would be changed in a step of execution of G, which obtained the publication of expression F. Each

F's publication V_F is bound to variable x, and do G with the value of x, then append this publication $V_{G(x)}$ at the end of sequence $\boldsymbol{S}$.

$$\mathcal{N}_{2.1} \triangleq \boldsymbol{S}' = \boldsymbol{S} \circ \langle V_{G(x)} \rangle \wedge x = V_F \tag{5}$$

To express liveness $\mathcal{L}_{2.1}$, the specification is strengthened to the form, which means that a step must eventually occur if $\mathcal{N}_{2.1}$ is *continuously* enabled.

$$\mathcal{L}_{2.1} \triangleq \mathbf{WF}_S(\mathcal{N}_{2.1}) = \Box(\Box ENABLED \ \ \langle \mathcal{N}_{2.1} \rangle_S \Rightarrow \Diamond \langle \mathcal{N}_{2.1} \rangle_S) \tag{6}$$

Mixing up the formulas (4)–(6), we get formal specification of sequential expression F >x> G. Hence, its semantics ***Seq1*** defined by

Seq1 : $Init_{2.1} \wedge \Box[\mathcal{N}_{2.1}]_S \wedge \mathcal{L}_{2.1}$ ***Semantics 2.1***

- **F >P> G**

The sequential combinator may be written as F >P> G, where P is a pattern instead of just a variable name. Any value published by F is matched against the pattern P. If this match is successful, a new execution of G begins, with all of the bindings from the match [25].

Example 3.3. In expression *((false,1)|(true,2)|(true,3)|(false,4)) >(true,y)> power(y)*, two expressions *(true,2)* and *(true,3)* can be successfully matched with pattern *(true,y)*, then the variable y could be bound to 2 and 3 in ether order. Therefore, the entire expression published 4 and 9 in either order.

Supposing publications of F >P> G are stored in a sequence $\boldsymbol{S}$, the initial predicate $Init_{2.2}$ is straightforward.

$$Init_{2.2} \triangleq \boldsymbol{S} = \emptyset \tag{7}$$

The next-relation $\mathcal{N}_{2.2}$ that specifies how the value of $\boldsymbol{S}$ would be changed in a step of execution of G, which obtained the successfully matched publication of expression F. If match is successful $P(V_F) = TRUE$, and do G with this value V_F, then append this publication $V_{G(V_F)}$ at the end of sequence $\boldsymbol{S}$. On the other hand, if $P(V_F) = FALSE$, sequence $\boldsymbol{S}$ is not changed.

$$\begin{aligned} \mathcal{N}_{2.2} \triangleq \mathbf{CASE} \ \ & P(V_F) = FALSE \rightarrow \boldsymbol{S}' = \boldsymbol{S} \\ & \blacksquare P(V_F) = TRUE \rightarrow \boldsymbol{S}' = \boldsymbol{S} \circ \langle V_{G(V_F)} \rangle \end{aligned} \tag{8}$$

To express liveness $\mathcal{L}_{2.2}$, the specification is strengthened to the form, which means that a step must eventually occur if $\mathcal{N}_{2.2}$ is *continuously* enabled.

$$\mathcal{L}_{2.2} \triangleq \mathbf{WF}_S(\mathcal{N}_{2.2}) = \Box(\Box ENABLED \ \ \langle \mathcal{N}_{2.2} \rangle_S \Rightarrow \Diamond \langle \mathcal{N}_{2.2} \rangle_S) \tag{9}$$

Mixing up the formulas (7)–(9), we get formal specification of sequential expression F >P> G. Hence, its semantics ***Seq2*** defined by

$\boldsymbol{Seq2} : Init_{2.2} \wedge \Box[\mathcal{N}_{2.2}]_S \wedge \mathcal{L}_{2.2}$ ***Semantics 2.2***

- **F >> G**

This is equivalent to using a wildcard pattern: F $> - >$ G. Every publication of F will match the wildcard pattern "–", causing an execution of G for every individual publication of F. No bindings will be made in G from these publications [25].

Example 3.4. Expression *(add(1,2)|sub(6,1)) >> power(10)* published 100 twice.

Supposing publications of F >> G are stored in a sequence $\boldsymbol{S}$, the initial predicate $Init_{2.3}$ is straightforward.

$$Init_{2.3} \triangleq \boldsymbol{S} = \emptyset \tag{10}$$

The next-relation $\mathcal{N}_{2.3}$ specifies appending value of G at the end of $\boldsymbol{S}$ after getting every publication from expression F, which is described by using $isPub(V_F) = TRUE$.

$$\mathcal{N}_{2.3} \triangleq \textbf{CASE} \;\; isPub(V_F) = TRUE \rightarrow \boldsymbol{S}' = \boldsymbol{S} \circ \langle V_G \rangle \tag{11}$$

To express liveness $\mathcal{L}_{2.3}$, the specification is strengthened to the form, which means that a step must eventually occur if $\mathcal{N}_{2.3}$ is *continuously* enabled.

$$\mathcal{L}_{2.3} \triangleq \mathbf{WF}_S(\mathcal{N}_{2.3}) = \Box(\Box ENABLED \;\; \langle \mathcal{N}_{2.3} \rangle_S \Rightarrow \Diamond \langle \mathcal{N}_{2.3} \rangle_S) \tag{12}$$

Mixing up the formulas (10)–(12), we get formal specification of sequential expression F >> G. Hence, its semantics ***Seq3*** defined by

$\boldsymbol{Seq3} : Init_{2.3} \wedge \Box[\mathcal{N}_{2.3}]_S \wedge \mathcal{L}_{2.3}$ ***Semantics 2.3***

3.3 Semantics of Pruning Combinator

Parallel and sequential combinators described above can only spawn new computations, but never terminate an executing computation. Pruning combinator allows us to do just that. Syntax of pruning combinator is described as:

$$Prune ::= Expression < [Variable/Pattern] < Expression$$

Like the sequential combinator, there are also three types of pruning expressions.

$$\begin{array}{ll} F < x < G & \text{[where x is a variable]} \\ F < P < G & \text{[where P is a pattern]} \\ F << G & \end{array}$$

Those semantics are defined as following.

- **F <x< G**

In F <x< G, both F and G are started concurrently. Site calls are *lenient* in Orc programming language, which means a site call is running even when some of its parameters are not bound to values [1]. So, the site calls in F that have x as a parameters can proceed even though x is not bound to a value. When G publishes its first value, that value is bound to x in F, and then the execution of G is immediately killed. A killed expression cannot call any sites or publish any values. During the execution of F, any part of the execution that depends on x will be blocked until x is bound (to the first value published by G). If G never publishes a value, those parts remain blocked forever. The publications of F is the values published by the entire execution of expression F <x< G [25].

Example 3.5. In expression *power(x)<x<(add(1,2)|sub(6,1))*, the site call *power(x)* is made even though x is bound to a value, and the execution of *power(x)* would be suspend until x is bound. Both sites call of *add(1,2)* and *sub(6,1)* execute simultaneously. As soon as a value is received from either sites call, the value 3 or 5 is bound to x, the execution of *add(1,2)|sub(6,1)* is terminated, and *power(x)*'s execution resumes. Therefore, the entire expression published 9 or 25.

Supposing publications of F <x< G are stored in a sequence $\boldsymbol{S}$, the initial predicate $Init_{3.1}$ is straightforward.

$$Init_{3.1} \triangleq \boldsymbol{S} = \emptyset \tag{13}$$

The next-relation $\mathcal{N}_{3.1}$ that specifies how the value of $\boldsymbol{S}$ would be changed in a step of execution. The first G's publication V_G is bound to variable x, and do F with the value of x, then append this publication $V_{F(x)}$ at the end of sequence $\boldsymbol{S}$. If $\boldsymbol{S}$ is not empty, terminate expression G and F.

$$\begin{aligned}\mathcal{N}_{3.1} \triangleq \textbf{CASE} \;\; & S = \emptyset \rightarrow \boldsymbol{S}' = \boldsymbol{S} \circ \langle V_{F(x)} \rangle \wedge x = V_G \\ & \blacksquare S \neq \emptyset \rightarrow \boldsymbol{isTerminal}(G) = TRUE \\ & \qquad \wedge \boldsymbol{isTerminal}(F) = TRUE\end{aligned} \tag{14}$$

To express liveness $\mathcal{L}_{3.1}$, the specification is strengthened to the form, which means that a step must eventually occur if $\mathcal{N}_{3.1}$ is *continuously* enabled.

$$\mathcal{L}_{3.1} \triangleq \mathbf{WF}_S(\mathcal{N}_{3.1}) = \Box(\Box ENABLED \;\; \langle \mathcal{N}_{3.1} \rangle_S \Rightarrow \Diamond \langle \mathcal{N}_{3.1} \rangle_S) \tag{15}$$

Mixing up the formulas (13)–(15), we get formal specification of pruning expression F <x< G. Hence, its semantics ***Pru1*** defined by

Pru1 : $Init_{3.1} \wedge \Box[\mathcal{N}_{3.1}]_S \wedge \mathcal{L}_{3.1}$ ***Semantics 3.1***

- **F <P< G**

The pruning combinator may include a full pattern P instead of just a variable name. Any value published by G is matched against the pattern P. If this match is successful, then G is killed and all of the bindings of pattern P are made in F. Otherwise, the published value is simply ignored and G continues to execute [25].

Example 3.6. In *power(y) <(true,y)< ((false,1)|(true,2)|(true,3)|(false,4))*, two expressions *(true,2)* and *(true,3)* can be successfully matched with pattern *(true,y)*, but site call *power(y)* only use the first publication 2 or 3. Therefore, the entire expression published 4 or 9.

Supposing publications of F <P< G are stored in a sequence $\boldsymbol{S}$, the initial predicate $Init_{3.2}$ is straightforward.

$$Init_{3.2} \triangleq \boldsymbol{S} = \emptyset \tag{16}$$

The next-relation $\mathcal{N}_{3.2}$ that specifies how the value of $\boldsymbol{S}$ would be changed in a step of execution of F, which obtained the first successfully matched publication of expression G, expressed by $P(V_G) = TRUE$. The first successfully matched publication V_G is bound to variable x, and do F with the value of x, then append this publication $V_{F(x)}$ at the end of sequence $\boldsymbol{S}$. If $\boldsymbol{S}$ is not empty, terminate expression G and F.

$$\begin{aligned} \mathcal{N}_{3.2} \triangleq \textbf{CASE} \ \ & S = \emptyset \wedge P(V_G) = FALSE \rightarrow \boldsymbol{S}' = \boldsymbol{S} \\ \blacksquare & S = \emptyset \wedge P(V_G) = TRUE \rightarrow \boldsymbol{S}' = \boldsymbol{S} \circ \langle V_{F(V_G)} \rangle \\ \blacksquare & S \neq \emptyset \rightarrow \boldsymbol{isTerminal}(G) = TRUE \\ & \wedge \boldsymbol{isTerminal}(F) = TRUE \end{aligned} \tag{17}$$

To express liveness $\mathcal{L}_{3.2}$, the specification is strengthened to the form, which means that a step must eventually occur if $\mathcal{N}_{3.2}$ is *continuously* enabled.

$$\mathcal{L}_{3.2} \triangleq \textbf{WF}_S(\mathcal{N}_{3.2}) = \Box(\Box ENABLED \ \ \langle \mathcal{N}_{3.2} \rangle_S \Rightarrow \Diamond \langle \mathcal{N}_{3.2} \rangle_S) \tag{18}$$

Mixing up the formulas (16)–(18), we get formal specification of pruning expression F <P< G. Hence, its semantics ***Pru2*** defined by

$\boldsymbol{Pru2}$: $Init_{3.2} \wedge \Box[\mathcal{N}_{3.2}]_S \wedge \mathcal{L}_{3.2}$ ***Semantics 3.2***

- **F << G**

This is equivalent to using a wildcard pattern, F < _ < G. G continues to execute until it publishes a value. Any value published by G will match the wildcard pattern "_". After the successful match, G is killed, but no bindings are made in F. No part of execution of F is suspended by the pruning combinator since there is no variable to be bound [25].

Example 3.7. Expression *power(10) << (add(1,2)|sub(6,1))* published one value 100.

Supposing publications of F << G are stored in a sequence $\boldsymbol{S}$, the initial predicate $Init_{3.3}$ is straightforward.

$$Init_{3.3} \triangleq \boldsymbol{S} = \emptyset \tag{19}$$

The next-relation $\mathcal{N}_{3.3}$ that specifies specifies appending value of F at the end of $\boldsymbol{S}$ after getting first publication from expression G. If $\boldsymbol{S}$ is not empty, terminate expression G and F.

$$\begin{aligned}\mathcal{N}_{3.3} \triangleq \textbf{CASE}\ \ S = \emptyset \wedge isPub(V_G)\ &=\ TRUE \rightarrow \boldsymbol{S'} = \boldsymbol{S} \circ \langle V_F \rangle \\ &\blacksquare S \neq \emptyset \rightarrow \boldsymbol{isTerminal}(G) = TRUE \\ &\wedge \boldsymbol{isTerminal}(F) = TRUE\end{aligned} \quad (20)$$

To express liveness $\mathcal{L}_{3.3}$, the specification is strengthened to the form, which means that a step must eventually occur if $\mathcal{N}_{3.3}$ is *continuously* enabled.

$$\mathcal{L}_{3.3} \triangleq \mathbf{WF}_S(\mathcal{N}_{3.3}) = \Box(\Box ENABLED\ \ \langle \mathcal{N}_{3.3} \rangle_S \Rightarrow \Diamond \langle \mathcal{N}_{3.3} \rangle_S) \quad (21)$$

Mixing up the formulas (19)–(21), we get formal specification of pruning expression F << G. Hence, its semantics ***Pru3*** defined by

$\boldsymbol{Pru3} : Init_{3.3} \wedge \Box[\mathcal{N}_{3.3}]_S \wedge \mathcal{L}_{3.3}$ ***Semantics 3.3***

3.4 Semantics of Otherwise Combinator

Syntax of otherwise combinator is described as:

$$Otherwise ::= Expression; Expression$$

Otherwise combinator exploits halting of expressions. In F ; G, execution of F is first started. If F halts, and has not published any value, then G executes. If F publishes one or more values, then G is ignored. The publications of F ; G are those of F if F publishes, or those of G if F is silent [25].

Let us introduce an auxiliary sequence $\boldsymbol{S}_F$, storing the publications of F, and a sequence $\boldsymbol{S}_G$, storing the publications of G. Supposing publications of F ; G are stored in a sequence $\boldsymbol{S}$, we defined its initialization.

$$Init_4 \triangleq \boldsymbol{S} = \emptyset \wedge \boldsymbol{S}_F = \emptyset \wedge \boldsymbol{S}_G = \emptyset \quad (22)$$

The next-relation $\mathcal{N}_4$ that specifies that how the value of $\boldsymbol{S}$ would be changed in a step of execution of F or G. If $\boldsymbol{S}_F$ is not empty, $\boldsymbol{S}$ store publication of F, otherwise, it store publication of G.

$$\begin{aligned}\mathcal{N}_4 \triangleq \textbf{CASE}\ \ &\boldsymbol{S}_F \neq \emptyset \wedge isTerminal(F) = TRUE \rightarrow \boldsymbol{S'} = \boldsymbol{S}_F \\ \blacksquare\ &\boldsymbol{S}_F = \emptyset \wedge isTerminal(F) = TRUE \rightarrow \boldsymbol{S'} = \boldsymbol{S}_G\end{aligned} \quad (23)$$

To express liveness $\mathcal{L}_4$, the specification is strengthened to the form, which means that a step must eventually occur if $\mathcal{N}_4$ is *continuously* enabled.

$$\mathcal{L}_4 \triangleq \mathbf{WF}_S(\mathcal{N}_4) = \Box(\Box ENABLED\ \ \langle \mathcal{N}_4 \rangle_S \Rightarrow \Diamond \langle \mathcal{N}_4 \rangle_S) \quad (24)$$

Mixing up the formulas (22)–(24), we get formal specification of otherwise expression F ; G. Hence, its semantics ***Oth*** defined by

$\boldsymbol{Oth} : Init_4 \wedge \Box[\mathcal{N}_4]_S \wedge \mathcal{L}_4$ ***Semantics 4***

4 Dining Philosophers Problem

Dining Philosophers Problem was originally introduced for a ring topology by Dijkstra [26]. Five philosophers are sitting around a circular table. Each philosopher has his own place, a single fork between each pair of adjacent philosophers. Any philosopher may decide to eat at any time and requires both of his forks to do so. Orc program [27] of dining philosophers problem is given as following.

```
1: def shuffle(a,b) = if (Random(2) = 1) then (a,b) else (b,a)
   – Randomly swap two elements
2: def take((a,b)) = a.acquire() >> b.acquireD() ; a.release() >> take(shuffle(a,b))
   – Acquire two forks in the order given
3: def drop(a,b) = (a.release(), b.release()) >> signal
   – Release two forks
4: def phil(a,b,name) =
       def thinking() = Rwait(Random(1000))
       def hungry() = take((a,b))
       def eating() = Println(name + "is eating.") >>
            Rwait(Random(1000)) >>
            Println(name + "has finished eating.") >> drop(a,b)
       thinking() >> hungry() >> eating() >> phil(a,b,name)
   – Start a philosopher process with forks a and b
5: def dining(n) =
       val forks = Table(n+1, lambda(_) = Semaphore(1))
       def phils(0) = stop
       def phils(i) = phil(forks(i), forks((i+1)% n), "Philosopher" + i) | phils(i-1)
       phils(n)
   – Start n philosophers dining in a ring
6: Let( dining(5) | Rwait(10000) ) >> Println("Simulation stopped.") >> stop
   – Simulate 5 philosophers for 10 seconds before halting
```

4.1 Semantics of Take Expression

Firstly, let us define specification of the 2nd definition in above program by using our semantics. ***take((a,b))*** is defined as the following expression ***TakeExp***.

- ***TakeExp***: a.acquire() >> b.acquireD() ; a.release() >> take(shuffle(a,b))

According *Combinator Precedence Level*, given in Sect. 2.1, sequential combinator has higher precedence over otherwise combinator. The above expression can be divided into two subexpressions, ***TakeSubExp1*** and ***TakeSubExp2***.

- ***TakeSubExp1***: a.acquire() >> b.acquireD()

Supposing sequence $\boldsymbol{S}_1$ is used to store the publication of ***TakeSubExp1***, so its initial predicate is $\boldsymbol{S}_1 = \emptyset$, and the next-relation $\mathcal{N}_{TakeSubExp1}$ is defined.

$$\mathcal{N}_{TakeSubExp1} \triangleq \textbf{CASE} \;\; isPub(V_{a.acquire()}) = TRUE \rightarrow \boldsymbol{S}_1' = \boldsymbol{S}_1 \circ \langle V_{b.acquireD()} \rangle \tag{25}$$

- ***TakeSubExp2***: a.release() >> take(shuffle(a,b))

Similarly, supposing sequence $\boldsymbol{S}_2$ is used to store the publication of ***TakeSubExp2***, so its initial predicate is $\boldsymbol{S}_2 = \emptyset$, and the next-relation $\mathcal{N}_{TakeSubExp2}$ is defined.

$$\mathcal{N}_{TakeSubExp2} \triangleq \textbf{CASE} \;\; isPub(V_{a.release()}) = TRUE \rightarrow \boldsymbol{S}_2' = \boldsymbol{S}_2 \circ \langle V_{take(shuffle(a,b))} \rangle \tag{26}$$

- ***TakeExp***: ***TakeSubExp1*** ; ***TakeSubExp2***

Supposing publications of ***TakeExp*** are stored in a sequence $\boldsymbol{S}$, we write an initial predicate $Init_{TakeExp}$.

$$Init_{TakeExp} \triangleq \boldsymbol{S}_1 = \emptyset \wedge \boldsymbol{S}_2 = \emptyset \wedge \boldsymbol{S} = \emptyset \tag{27}$$

The next-relation $\mathcal{N}_{TakeExp}$ is defined.

$$\begin{aligned} \mathcal{N}_{TakeExp} \triangleq \textbf{CASE} \;\; & \boldsymbol{S}_1 \neq \emptyset \wedge isTerminal(TakeSubExp1) = TRUE \rightarrow \boldsymbol{S}' = \boldsymbol{S}_1 \\ \blacksquare \;\; & \boldsymbol{S}_1 = \emptyset \wedge isTerminal(TakeSubExp2) = TRUE \rightarrow \boldsymbol{S}' = \boldsymbol{S}_2 \end{aligned} \tag{28}$$

4.2 Semantics of Philosopher Expression

Then we define specification of each philosopher using our semantics. ***phil(a,b, name)*** is defined as the following expression ***PhilExp***.

- ***PhilExp***: thinking() >> hungry() >> eating() >> phil(a,b,name)

This recursive expression can be divided into two subexpressions, ***PhilSubExp1*** and ***PhilSubExp2***.

- ***PhilSubExp1***: thinking() >> hungry()

Supposing sequence $\boldsymbol{R}_1$ is used to store the publication of ***PhilSubExp1***, so its initial predicate is $\boldsymbol{R}_1 = \emptyset$, and the next-relation $\mathcal{N}_{PhilSubExp1}$ is defined.

$$\mathcal{N}_{PhilSubExp1} \triangleq \textbf{CASE} \;\; isPub(V_{thing()}) = TRUE \rightarrow \boldsymbol{R}_1' = \boldsymbol{R}_1 \circ \langle V_{hungry()} \rangle \tag{29}$$

- ***PhilSubExp2***: ***PhilSubExp1*** >> eating()

Similarly, supposing sequence $\boldsymbol{R}_2$ is used to store the publication of ***PhilSubExp2***, so its initial predicate is $\boldsymbol{R}_2 = \emptyset$, and the next-relation $\mathcal{N}_{PhilSubExp2}$ is defined.

$$\mathcal{N}_{PhilSubExp2} \triangleq \textbf{CASE} \;\; \boldsymbol{R}_1 \neq \emptyset \rightarrow \boldsymbol{R}_2' = \boldsymbol{R}_2 \circ \langle V_{eating()} \rangle \tag{30}$$

PhilExp is recursive expression, so it can be rewrote like this.

- ***PhilExp***: ***PhilSubExp2*** >> ***PhilExp***

Supposing publications of ***PhilExp*** are stored in a sequence $\boldsymbol{R}$, and the number of ***PhilExp***'s publications are stored in a integer variable ***num***, and its initial state is $\boldsymbol{R} = \emptyset \wedge \boldsymbol{num} = 0$. We defined its next-relation $\mathcal{N}_{PhilExp}$.

$$\mathcal{N}_{PhilExp} \triangleq \textbf{CASE} \;\; \boldsymbol{R}_2 \neq \emptyset \rightarrow \boldsymbol{R}' = \boldsymbol{R} \circ \boldsymbol{R}_2 \wedge num' = num + 1 \tag{31}$$

4.3 Safety

In the verification of concurrent programs two kinds of properties are of primary importance: *safety properties*, asserting that something bad never happens, and *liveness properties* asserting that something good will eventually happen. Two popular safety properties are generally considered: *invariance* and *deadlock freedom.*

According to the above Orc program of dining philosophers, we know the i^{th} philosophers has two forks $fork_i$ and $fork_{(i+1)\%n}$. Supposing $Phil_0$ is equal to $Phil_n$, the i^{th} fork shared by two philosophers $Phil_i$ and $Phil_{i-1}$.

An invariance property is expressed by a TLA$^+$ formula $\Box P$, where P is a predicate. In expression $forks = Table(n+1, \textbf{lambda}(\_) = Semaphore(1))$, each fork is defined by using semaphore with initial value one. So it implies the following invariant property: "*Every fork can't be used by two philosophers*", and it is defined as ***ForkSpe***.

$$ForkSpe \triangleq \Box(\forall i : 1 \leq i \leq n : \neg((fork_i \in Phil_i) \wedge (fork_i \in Phil_{i-1})) \tag{32}$$

Each philosopher could be eating only when he (or she) is hungry, and has successfully got both forks. This condition can be expressed as following.

$$((fork_i \in Phil_i) \wedge (fork_{(i+1)\%n} \in Phil_i)) \Rightarrow isEating(Phil_i) \tag{33}$$

So, let us consider the second invariant property: "*Two neighbor philosophers not eat simultaneously*", and it is defined as ***PhilMutual***.

$$\begin{aligned} PhilMutual \triangleq\; & ForkSpe \Rightarrow \Box(\forall i : 1 \leq i \leq n : \\ & \neg(isEating(Phil_i) \wedge isEating(Phil_{(i+1)\%n})) \end{aligned} \tag{34}$$

Deadlock occurs when two or more philosophers in a system are blocked forever, because of requirements that can never be satisfied. There is a deadlock when each philosopher got left fork, and is waiting right fork. But it is free of deadlock in this program.

Expressions ***shuffle(a,b)*** and ***take((a,b))*** could effectively avoid deadlock. If $Random(2) = 1$, the return tuple of expression ***shuffle(a,b)*** is (a, b), which means firstly getting a, then getting b; otherwise, its return tuple is (b, a), which means firstly getting b, then getting a. If a philosopher got one fork, but failed to get another fork in a given period of time, the program entered into a mediate state, called ***GotOneFork*** $= (fork_i \in Phil_i) \oplus (fork_{(i+1)\%n} \in Phil_i)$, where $\oplus$ is XOR operator. At the moment, the philosopher release this fork, and try to get two forks in the same order or different order, depending on the value of $Random(2) = 1$.

Supposing a sequence P_i is storing the order of fork for i^{th} philosophers. The specification of philosopher ***PhilSpe*** is expressed as follows.

$$\begin{aligned} PhilSpe \triangleq \Box(\forall i : 1 < i < 5 : \textbf{CASE}\ \ & Random(2) = 1 \wedge (fork_i \in Phil_i) \\ & \wedge(fork_{(i+1)\%n} \in Phil_i) \rightarrow P_i = \langle i, (i+1)\%n \rangle \\ & \blacksquare Random(2) \neq 1 \wedge (fork_{(i+1)\%n} \in Phil_i) \\ & \wedge(fork_i \in Phil_i) \rightarrow P_i = \langle (i+1)\%n, i \rangle \\ & \blacksquare (fork_i \in Phil_i) \oplus (fork_{(i+1)\%n} \in Phil_i) \rightarrow \\ & repeat(take(shuffle(fork_i, fork_{(i+1)\%n}))) \end{aligned} \tag{35}$$

Mixing up the above specifications, it's easier to express free-deadlock ***DeadlockFree*** in the following property.

$$DeadlockFree \triangleq ForkSpe \wedge PhilMutual \wedge PhilSpe \tag{36}$$

4.4 Liveness

A liveness property is that if a philosopher thinks and want to eat, he (or she) eventually would eat. Let us remark a expression ***PhilSubExp2***: *PhilSubExp1* >> eating(), where ***PhilSubExp1***: thinking() >> hungry(). Its next-relation formula $\mathcal{N}_{TakeSubExp2}$ is discussed in Sect. 4.2. Hence, we enhance this formula by a weak fairness condition for ***PhilSubExp2*** expression.

$$\begin{aligned} \mathbf{WF}_{\langle R_1,R_2 \rangle}(\mathcal{N}_{PhilSubExp2}) = \Box(\Box ENABLED\ \ & \langle \mathcal{N}_{PhilSubExp2} \rangle_{\langle R_1,R_2 \rangle} \\ & \Rightarrow \Diamond \langle \mathcal{N}_{PhilSubExp2} \rangle_{\langle R_1,R_2 \rangle}), \end{aligned} \tag{37}$$

where sequences R_1 and R_2 are also defined in Sect. 4.2.

The weak fairness $\mathbf{WF}_{\langle R_1,R_2 \rangle}(\mathcal{N}_{PhilSubExp2})$ means that a step must eventually occur if $\mathcal{N}_{PhilSubExp2}$ is *continuously* enabled, so hungry philosopher would eat eventually.

4.5 Comparison

A new TLA-based model for specifying and verifying concurrent programs is proposed in paper [28], it combinated ADT and TLA, and illustrated dining philosophers problem. In Palmer's doctoral-dissertation [29], he give a TLA$^+$ specification of the classic dining philosophers problem, it defined formula about three states (including "Thinking", "Got One Fork" and "Eating") in the life of a philosopher. Compared with them, our semantics of program, and our description of safety and liveness could be simpler.

5 Related Works

Theretofore, several attempts have been made to find semantics of Orc. In 2004, a tree semantics of three combinators (parallel, sequential and pruning combinator) has been defined by Tony Hoare and Jayadev Misra [20], its main achievement is to permit simple proofs of familiar algebraic identities that hold between programs with different syntactic forms. Based on labeled transaction system, Misra and Cook also defined an operational semantics for three kinds of combinators [2]. Trace-based denotational semantics model without time has also been demonstrated in [21], this model induced a congruence on Orc programs and facilitates reasoning about them. A partial-order semantics [22] was defined using heaps, which are then easily translated into asymmetric event structures. A timed operational semantics of Orc reasoning about delays, and a denotational semantics of Orc program with a set of traces, was presented in paper [23]; and authors also showed the two semantics are equivalent. A denotational semantical model for Orc language is proposed by Jifeng He [24], and this model gives the same semantical interpretation to the implementations and specifications of Orc program. In the latest Jayadev Misra's book [1], he summarized two parts of Orc semantics, one is asynchronous semantics without the notion of real time, another is synchronous semantics with real time.

Compared with the above semantics of Orc, our formal semantics of Orc combinators pay more attention to its behavior, expressed by next-state relation, and its liveness properties demonstrated by using weak fairness of TLA$^+$.

6 Conclusions

Concurrency will continue to permeate many areas, including client-service systems, transaction processing system and web services. *Orc theory*, consists of Orc calculus and Orc language, provide high-level constructs suitable for structured concurrent programming. Its four powerful combinators is inherently concurrent, which means each combinator captures a different feature of concurrency.

The motivation of the paper is to extend Orc concurrent mechanism in our abstract sequential programming language, called *Apla*. Hence, we have presented formal semantics for Orc expression, combined by combinators. Our main work is to elaborate behavior and liveness property of Orc combinators by using TLA$^+$ specification, consisting of initial states, next-state relation, and fairness requirement. Finally, a typical dining philosophers problem is illustrated application of our semantics.

After understanding the semantics of Orc combinators, we will continue research on implementation of Orc concurrent mechanism in our *Apla* language, so that it becomes sequential and concurrent language $Apla^{+}$; and then its supporting toolkit $Apla^{+}$ *ToJava* generator, automatically translating abstract concurrent $Apla^{+}$ program into Java concurrent program, will be designed in the near future.

Acknowledgments. The authors thanks Professor Jayadev Misra, inventor of Orc theory, for discussion about Orc combinators and his tutorial lessons in FACS2013.

References

1. Misra, J.: Structured Concurrent Programming (2013). http://www.cs.utexas.edu/users/misra/temporaryFiles.dir/Orc.pdf
2. Misra, J., Cook, W.R.: Computation orchestration: a basis for wide-area computing. J. Softw. Syst. Model. **6**(83), 83–110 (2007)
3. Kitchin, D., Quark, A., Cook, W., Misra, J.: The Orc programming language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 1–25. Springer, Heidelberg (2009)
4. Kitchin, D.: Orchestration and atomicity. Ph.D. dissertation, The University of Texas at Austin, August (2013)
5. Orc Language Project (2014). http://orc.csres.utexas.edu/index.shtml
6. Jayadev, M.: Structured orchestration of data and computation. In: Keynote-speeach in 10th International Symposium on Formal Aspects of Component Software, 28–30 October 2013, Nanchang, China (2013). http://www.cs.utexas.edu/users/misra/FACS.pdf
7. Ben-Ari, M.: Mathematical Logic for Computer Science, 2nd edn. Springer, New York (2001)
8. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on the Foundations of Computer Science, pp. 46–57. IEEE (1977)
9. Lamport, L.: Specifying concurrent program modules. ACM Trans. Program. Lang. Syst. **5**(2), 190–222 (1983)
10. Lamport, L.: The temporal logic of actions. Research report 79, digital equipment corporation, systems research center. To appear in transactions on programming language and systems (1991)
11. Lamport, L.: Specifying Systems: The TLA^{+} Language and Tools for Hardware and Software Engineers. Addison-Wesley, Reading (2003)
12. Lamport, L.: Verification and specification of concurrent programs. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) A Decade of Concurrency Reflections and Perspectives. LNCS, vol. 803, pp. 347–374. Springer, Heidelberg (1994)
13. Jinyun, X.: An Abstract Programming Language Apla. Report of Jiangxi Normal University (2001)
14. Jinyun, X.: A unified approach for developing efficient algorithm of programs. J. Comput. Sci. Technol. **12**(4), 314–329 (1997)
15. Jinyun, X.: A practicable approach for formal development of algorithmic programs. In: Proceeding of the International Symposium on Future software Technology, Nanjing, China (1999)
16. Jinyun, X.: PAR method and its supporting platform. In: Proceeding of the 1st Asian Working Conference on Verified Software (AWCVS 2006), pp. 29–31 (2006)

17. 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)
18. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Sci. Comput. Program. **70**(1), 31–61 (2008)
19. Tian, C., Duan, Z.: Expressiveness of propositional projection temporal logic with star. Theor. Comput. Sci. **412**, 1729–1744 (2011)
20. Hoare, T., Menzel, G., Misra, J.: A tree semantics of an orchestration language. Lecture Notes for NATO summer school, Marktoberdorf (2004). http://orc.csres.utexas.edu/papers/Semantics.Orc.pdf
21. Kitchin, D.E., Cook, W.R., Misra, J.: A language for task orchestration and its semantic properties. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 477–491. Springer, Heidelberg (2006)
22. Rosario, Sidney, Kitchin, David E., Benveniste, Albert, Cook, William, Haar, Stefan, Jard, Claude: Event structure semantics of Orc. In: Dumas, Marlon, Heckel, Reiko (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 154–168. Springer, Heidelberg (2008)
23. Wehrman, I., Kitchin, D., Cook, W.R., Misra, J.: Timed semantics of orc. Theor. Comput. Sci. **402**(2–3), 234–248 (2008)
24. Li, Q., Zhu, H., He, J.: A denotational semantical model for orc language. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, pp. 106–120. Springer, Heidelberg (2010)
25. Orc Reference Manual v2.1.0 (2013). http://orc.csres.utexas.edu/documentation/html/refmanual/index.html
26. Dijkstra, E.W.: Hierarchical ordering of sequenial processes. In: Operating Systems Techniques. Academic Press, New York (1971)
27. Try Orc! A example of Dining Philosophers (2014). http://orc.csres.utexas.edu/tryorc.shtml#tryorc/small-demos/philosopher.orc
28. Slimani, Y., Dahoy, E.H.: Logic Abstract Modules: A new TLA-based model for Specifying and Verifying Concurrent Programs. Department Informatique, Faculte del Scineces de Tunis (1998). www.di.unipi.it/brogi/ResearchActivity/COCL98/Papers/p6.ps
29. Palmer, R.L.: Formal anaysis for MPI-based high performance computing software. Doctoral Dissertation, The University of Utach (2007)

Incremental Points-to Analysis for Java via Edit Propagation

Yuting Chen[1(✉)], Qiuwei Shi[1,2], and Weikai Miao[2]

[1] School of Software, Shanghai Jiao Tong University, Shanghai 200240, China
chenyt@cs.sjtu.edu.cn

[2] School of Software, East China Normal University, Shanghai 200062, China
louisxivr@outlook.com, wkmiao@sei.ecnu.edu.cn

Abstract. Points-to analysis is a static analysis technique which computes the relationships between the program variables and the heap references. It has been widely used in program optimization, program understanding, and error detection. Inclusion-based points-to analysis computes the points-to sets in a program by translating the program into a set of inclusion constraints on the points-to sets and then solving them to yield the desired results. Yet the analysis faces a difficulty in that a program can be frequently changed in its development, and great efforts may be exhausted to re-generate the inclusion constraints and re-solve them. In this paper, we extend the inclusion-based points-to analysis to an incremental one called *Inc-PTA*. The essential idea of Inc-PTA is to sum up the program changes into an editscript of a sequence of successive edits, and then to propagate the edits to the constraints followed by taking a demand-driven points-to analysis of the program. We also discuss about the correctness of Inc-PTA, and believe that Inc-PTA can provide with a cost-effective solution to incremental points-to analysis.

Keywords: Constraint solving · Incremental Points-to Analysis · Edit propagation

1 Introduction

Points-to analysis is a static analysis technique which computes the relationships between the program variables and the heap references [1,2]. Inclusion-based points-to analysis (i.e., Andersen-style points-to analysis [2]) is a classical points-to analysis technique. It advocates an idea of translating a program into a set of inclusion constraints on the points-to sets and then iteratively solving these constraints to yield the results [2–4].

Inclusion-based points-to analysis has been widely used in program optimization, program optimization, program understanding, and error detection [5]. Meanwhile its efficiency is always a concern in practice. One main obstacle is that a large number of strongly-coupled constraints may be produced during analysis, solving of which are usually iterated, and at each iteration most, if not all,

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 164–178, 2015.
DOI: 10.1007/978-3-319-17404-4_11

constraints need to participate in the solving [3,4]. In addition, some system and/or user libraries are usually included in the analysis, which requires the massive resources be consumed during analysis because the reachable objects and methods can be numerous [6]. Although it is often conducted to ease the difficulty by performing the sparse analysis or reducing the domains of variables of interest [7,8], resource-intensive remains to be the natural instincts of points-to analysis.

On the other side, a software system in development can be frequently changed, making points-to analysis tedious. Efforts must be made in order to alleviate the difficulty and take the points-to analysis in an incremental style, which refrains the points-to information from being recomputed from scratch, but achieves them by updating the points-to sets that have been previously solved. Although some solutions (e.g., [9]) have been proposed to seek to perform efficient incremental points-to analysis, they do not harmonize well with the commonly-used inclusion-based analysis algorithms in that the inclusion constraints may not be updated and re-solved. One key to these solutions is to focus on " reachability" [10], which requires the engineers scrupulously identify the change points in the program to be analyzed and then update the points-to sets of the variables that are reachable to the change points. A path tracing technique (e.g., slicing) is usually adopted to derive the reachability information from the program.

In this paper, we extend the inclusion-based points-to analysis for Java to an incremental one called *Inc-PTA* (Incremental Points-to Analysis). Inc-PTA is motivated by an insight of "reusing and re-solving the constraints". Let `Prog` and `Prog'` be the versions of a Java program before and after change, respectively. Let the sets of inclusion constraints for `Prog` and `Prog'` be $Cons_{\texttt{Prog}}$ and $Cons_{\texttt{Prog'}}$, respectively. An observation can inevitably help find a fact that most inclusion constraints on `Prog` are repeated in those on `Prog'`, and solving of $Cons_{\texttt{Prog'}}$ also benefits from the process of solving of $Cons_{\texttt{Prog}}$ in that they endure the similar iterative computations.

1.1 Basic Approach

Inclusion-based points-to analysis defines and solves a set of inclusion constraints on the points-to relations. During analysis, a Java program `Prog` is at first transformed into the Jimple's three-address code [11]. After that, a PAG (Pointer Assignment Graph) is constructed to represent the inclusion constraints on `Prog` and the points-to sets are propagated throughout the graph until a fixed point is reached.

Inc-PTA is an incremental and change-adaptive approach to points-to analysis. It advocates the idea of reducing the efforts of re-generating and re-solving the constraints on `Prog'` by reusing and editing those on `Prog`. As Fig. 1 shows, once `Prog` is slightly changed to `Prog'`, an editscript describing how `Prog` is changed into `Prog'` is produced and then propagated to the points-to sets of `Prog`. We focus on two main issues in regard to the incremental points-to analysis for Java:

1. The intermediate representations of the program to be analyzed. We introduce two kinds of intermediate representations for supporting the incremental points-to analysis: *Inc-3AC* (Incremental Jimple's Three-Address Code)

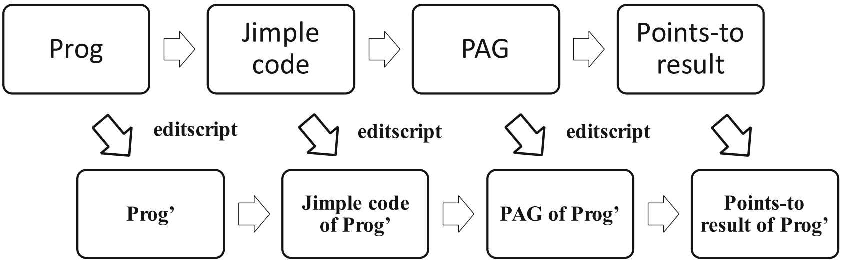


Fig. 1. Edit propagation during the incremental analysis.

representing the Jimple's code of a program along with all changes to the code, and *Inc-PAG* (Incremental Pointer Assignment Graph) representing the points-to relations given by a program along with all changes to them;
2. The edits and their propagations. We believe that edit propagation can be a key to incremental analysis: when the program is changed, the engineer summarizes the program changes into an editscript of a sequence of successive primitive edits followed by the propagation of the edits to the PAG. The points-to sets of the program can also be efficiently computed on the basis of the PAG and the edits on it.

1.2 Contributions

The paper makes two contributions:

1. We provide a lightweight solution to incremental points-to analysis which does not only hold the instincts for the traditional inclusion-based points-to analysis, but also adapt to the program changes in that it allows the program changes to be transferred throughout the analysis in an edit propagation manner.
2. We present three crucial activities of the incremental points-to analysis. The three activities focus on how the edits on the program are propagated to the Jimple's code, to the PAG, and to the points-to sets of the program, respectively. These activities supplement each other so that the points-to results for `Prog'` can be efficiently computed.

This paper is organized as follows: Sect. 2 briefly introduces the main idea of inclusion-based points-to analysis, Sect. 3 introduces the principle of Inc-PTA and make a discussion, Sect. 4 describes the related work, and Sect. 5 draws the conclusions and points out the future research directions.

2 Background

Inclusion-based points-to analysis defines and solves a set of inclusion constraints on the points-to relations. Lhoták has introduced PAG (Pointer Assignment Graph) as the internal representation of the program being analyzed [12]. A PAG is defined as a directed graph $< V, E >$ where

- V contains a set of vertices. A PAG uses four types of vertices (*allocation* vertices, *variable* vertices, *filed reference* vertices, and *concrete field* vertices) to represent the memory locations.
- E contains a set of edges. A PAG uses four types of edges, as next shows. Each edge (say $v_1 \rightarrow v_2$) represents an assignment of pointers from a source vertex v_1 to a sink one v_2, and is associated with an inclusion constraint on the points-to set of v_1 and that of v_2.
  1. An *allocation* edge (`L1:a = new C()` for example) is from an allocation vertex to a variable vertex. It represents an assignment of pointers to the objects represented by an allocation vertex to the location of the variable `a`. The inclusion constraint represented by the edge is $\{\texttt{L1}\} \subseteq pt(\texttt{a})$, where *pt* returns the points-to set of `a`.
  2. An *assignment* edge (`a = b` for example) is from a variable vertex to another one. It represents an assignment of pointers from the location of `b` to the location of `a`. A constraint represented by the edge is $pt(\texttt{b}) \subseteq pt(\texttt{a})$.
  3. A *load* edge (`a = b.f` for example) is an edge from a field reference vertex to a variable vertex. It represents a load of the appropriate field of some object (i.e., `b.f`) to the location of `a`. A constraint represented by the edge is $pt(\texttt{o.f}) \subseteq pt(\texttt{a})$ where $\texttt{o} \in pt(\texttt{b})$.
  4. A *store* edge (`b.f = a` for example) is from a variable vertex to a field reference vertex. It represents a store of `a` to the appropriate field reference. A constraint represented by the edge is $pt(\texttt{a}) \subseteq pt(\texttt{o.f})$ where $\texttt{o} \in pt(\texttt{b})$.

A PAG can be built by iterating through the Jimple input of the program and then creating the appropriate vertices and edges in the graph. For facilitating the incremental analysis, we assign each edge with a label list representing which lines of Jimple's code can contribute to the creation of the edge. For example, $v_1 \rightarrow_{0,1} v_2$ denotes that $v_1 \rightarrow v_2$ is created due to the instructions at lines 0 and 1.

After analysis, each vertex is assigned with a points-to set denoting the heap references pointed by the variable or the field represented by the vertex. The points-to analysis is performed by propagating the points-to sets throughout the PAG, i.e., solving the inclusion constraints on the points-to sets. The propagation is usually iterated along with the inclusion of the newly-discovered reachable methods and heap references into the PAG until a fixed point can be reached, i.e., the constraints are usually iteratively solved.

3 Principle of Inc-PTA

Inc-PTA advocates the idea of analyzing the points-to sets of `Prog'` by propagating the edits to the process of constraint solving and patching up the point-to sets of `Prog`. Some research questions may be raised during the incremental analysis, which will be answered in the remaining part of this section.

1. How is an edit be defined on a Java program to be analyzed?
2. For each edit on `Prog`, what kinds of edits need to be generated on its Jimple's code and PAG?
3. How are the constraints re-solved?
4. How is the correctness of the analysis guaranteed?

Listing 1.1. A BubbleSort program

```java
1  import java.util.ArrayList;
2  import java.util.Random;
3
4  public class Bubblesort {
5      public ArrayList<Integer> array;
6      public Integer sum;
7
8      public Bubblesort() {
9          array = new ArrayList<Integer>();
10         Random random = new Random();
11         for (int i = 0; i < 10; i++)
12             array.add(new Integer(random.nextInt(100)));
13         //+ sum^0=new Integer(0);
14     }
15
16     public int sort(boolean ascending) {
17         int k = 0;
18         while (k < array.size()) {
19             int t = k;
20             while (t > 0 && (ascending && array.get(t - 1) >~array.get(t) ||
21                 !ascending && array.get(t - 1) < array.get(t))) {
22                 swap(t - 1, t);
23                 t = t - 1;
24             }
25             //+ sum^1=new Integer(sum^2+array.get(k));
26             k = k + 1;
27         }
28         return array.size();
29     }
30
31     private void swap(int index1, int index2) {
32         Integer temp = array.get(index1);
33         array.set(index1, array.get(index2));
34         array.set(index2, temp);
35     }
36
37     public static void main(String[] args) {
38         Bubblesort bubblesort = new Bubblesort();
39         bubblesort.sort(false);
40     }
41 }
```

3.1 An Example

For facilitating the discussion, we take a Java program `BubbleSort` (see Listing 1.1) as an example. The program stores a set of random integers to an object of the ArrayList type, and then adopts the bubblesort algorithm to sort the integers in either an ascending or a descending order. `BubbleSort'` is an updated version of `BubbleSort` in that two statements (see lines 13 and 25) are inserted into the program for summing up all the integers in the array. In Listing 1.1 we take `sum` as the variable of interest and use $\texttt{sum}^i$ to denote the i^{th} appearance of `sum`.

For quickly moving into the essential idea of Inc-PTA, we omit the details of performing the inclusion-based points-to analysis of `BubbleSort`, but emphasize on how to perform the incremental analysis of `BubbleSort'`. Any engineer can use the SOOT's Spark points-to analysis framework [12] to construct the PAG of `BubbleSort` and compute the points-to sets.

3.2 Editscript

As an initial step, we summarize the program changes into an editscript. An editscript is composed of a sequence of primitive edits that are successively committed to the program. Some common types of primitive edits are enumerated as follows:

1. `Insert(Prog,`s_1`,`s_2`)`: insert a statement s_1 at a position after s_2;
2. `Delete(Prog,s)`: delete `s` from `Prog`;
3. `Update(Prog,`s_1`,`s_2`)`: update s_1 to s_2;
4. `Move(Prog,`s_1`,`s_2`)`: move s_1 to a position after s_2. The edit is functionally equivalent to `Delete(Prog,s)`•`Insert(Prog,`s_1`,`s_2`)`;
5. `Align(Prog,`s_1`,`s_2`)`: exchange the positions of s_1 and s_2;
6. `NOP(Prog)`: no operation on `Prog`.

An editscript is composed of a sequence of primitive edits. Here we use "•" to concatenate the primitive edits to form an editscript. For example, the changes of `BubbleSort` are summarized into an editscript $\texttt{EditScript}_{\texttt{BubbleSort}}$ of two primitive edits: an insertion of an initialization of the variable `sum` into `BubbleSort()`, and an insertion of a statement into the method `sort(boolean)` for summing up the elements in the array. Notice that in each edit we use the method signature (e.g., `sort(boolean)`) to indicate which method a statement is inserted into.

$\texttt{EditScript}_{\texttt{BubbleSort}}$:
`Insert(BubbleSort(),"13: sum`^{0}`=new Integer(0);")`
• `Insert(sort(boolean),"25: sum`^{1}`=new Integer(sum`^{2}`+`
`array.get(k));")`.

In the study, we use *ChangeDistiller*, a fine-grained source code change extraction tool [13,14], to recognize the edits on the programs to be analyzed. ChangeDistiller differs the ASTs (Abstract Syntax Trees) of `Prog` and `Prog'`, and produces an editscript that contains a minimal number of tree edit operations with which the AST of `Prog` can be edited into that of `Prog'`. The edits are formed into an editscript which provide us with precise information about the source code changes and as well the locations of the changes.

3.3 Edits on Jimple's Three Address Code

Points-to analysis is usually performed on the intermediate code of a program. Thus we also perform the incremental analysis of a program on its Jimple's three address code, but mark up in the Jimple's code the change information. Notice that although the SOOT optimization framework [15] can be used to generate the Jimple's code for `Prog'`, the traceability between the code and that of `Prog` is not easy to maintain since the programs may use different temporary variables and labels in their Jimple's code. Thus we take some extra steps to weave the edits on `Prog` into its Jimple's code (say $\texttt{Jimple}_{\texttt{Prog}}$) so that the resulting code (say $\texttt{Jimple}_{\texttt{Prog'}}$) can be traceable back to $\texttt{Jimple}_{\texttt{Prog}}$.

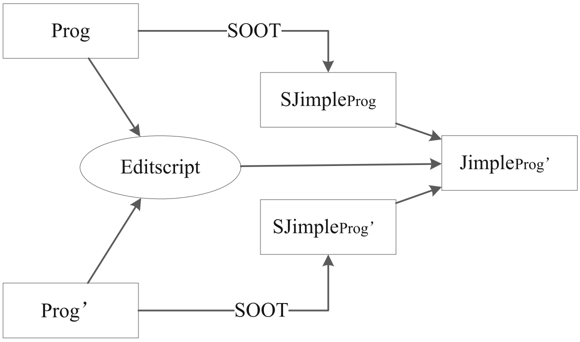


Fig. 2. Generation of the Jimple's code with change information for `Prog'`.

As Fig. 2 shows, we at first use the SOOT framework to generate the Jimple's code for both `Prog` and `Prog'`. Let the SOOT-generated Jimple's code of `Prog` and `Prog'` be $\texttt{SJimple}_{\texttt{Prog}}$ (same as $\texttt{Jimple}_{\texttt{Prog}}$) and $\texttt{SJimple}_{\texttt{Prog}'}$, respectively. We follow the next steps to generate $\texttt{Jimple}_{\texttt{Prog}'}$ that is traceable to $\texttt{Jimple}_{\texttt{Prog}}$:

- Print in $\texttt{SJimple}_{\texttt{Prog}}$ and $\texttt{SJimple}_{\texttt{Prog}'}$ a variety of tags to denote the relationships between the program constructs and the code segments. Each tag shows a program statement that contributes to the generation of an instruction, and it is printed on the line succeeding the instruction that it is attached to. With the tags, an engineer can determine the program statement that contributes to an instruction and, given a program statement, also seek the lines of Jimple's code it generates.
- Eliminate the labels and the `goto` instructions in the Jimple's code. Since the creation of a PAG edge relies on the existence of one or more instructions which hold the points-to relation of the edge but not their positions or the control flows in the code, we eliminate in the code the instruction labels and the `goto` instructions.
- Translate the editscript on `Prog` into an editscript on $\texttt{SJimple}_{\texttt{Prog}}$. We translate each edit on `Prog` to one or more edits on the Jimple's code. The patterns for translating the edits are given as follows.
  - $\texttt{Insert(Prog,}s_1\texttt{,}s_2\texttt{)}$ is transformed into insertion of $\texttt{SJimple}_{s1}$ into $\texttt{SJimple}_{\texttt{Prog}}$. $\texttt{SJimple}_{s1}$ represents the Jimple's code for s_1, and is obtained by seeking in $\texttt{SJimple}_{\texttt{Prog}'}$ the Jimple's instructions generated for s_1;
  - $\texttt{Delete(Prog,s)}$ is transformed into deletion of $\texttt{SJimple}_{s}$ from $\texttt{SJimple}_{\texttt{Prog}}$. $\texttt{SJimple}_{s}$ is obtained by seeking in $\texttt{SJimple}_{\texttt{Prog}}$ the Jimple's instructions generated for `s`;
  - $\texttt{Update(Prog,}s_1\texttt{,}s_2\texttt{)}$ is transformed into deletion of $\texttt{SJimple}_{s1}$ followed by insertion of $\texttt{SJimple}_{s2}$. Here $\texttt{SJimple}_{s1}$ and $\texttt{SJimple}_{s2}$ are obtained from $\texttt{SJimple}_{\texttt{Prog}}$ and $\texttt{SJimple}_{\texttt{Prog}'}$, respectively;
  - $\texttt{Move(Prog,}s_1\texttt{,}s_2\texttt{)}$ is transformed into moving $\texttt{SJimple}_{s1}$ to a position after $\texttt{SJimple}_{s2}$. Both $\texttt{SJimple}_{s1}$ and $\texttt{SJimple}_{s2}$ are obtained from $\texttt{SJimple}_{\texttt{Prog}}$;

```
1. Insert(BubbleSort(), "13: sum^0=new Integer(0);")⇒
   Insert(BubbleSort(), "+k0: $t0 = new java.lang.Integer;")
   •Insert(_, "+k1: specialinvoke $t0.<init>(0);")
   •Insert(_, "+k2: this.sum = $t0;")
2. Insert(sort(boolean),"25: sum^1=new Integer(sum^2 +array.get
   (k));")⇒
   Insert(sort(boolean), "+k3: $t0 = this.<Bubblesort:
   java.lang.Integer sum>;")
   •Insert(_, "+k4: $t1 = virtualinvoke $t0.<java.lang.Integer:
   int intValue()>();")
   •Insert(_, "+k5: $t2 = this.<Bubblesort: java.util.ArrayList
   array>;")
   •Insert(_, "+k6: $t3 = virtualinvoke $t2.<java.util.ArrayList:
   java.lang.Object get(int)>(k);")
   •Insert(_, "+k7: $t4 = (java.lang.Integer) $t3;")
   •Insert(_, "+k8: $t5 = virtualinvoke $t4.<java.lang.Integer:
   int intValue()>();")
   •Insert(_, "+k9: $t6 = $t1 + $t5;")
   •Insert(_, "+k10: $t7 = new java.lang.Integer;")
   •Insert(_, "+k11: specialinvoke $t7.<java.lang.Integer: void
   <init>(int)>($t6);")
   •Insert(_, "+k12: this.<Bubblesort: java.lang.Integer sum> =
   $t7;")
```

Fig. 3. Transformation of the edits on BubbleSort into the edits on its Jimple's code.

- Align(Prog,s$_1$,s$_2$) is transformed into exchanging the positions of SJimple$_{s1}$ and SJimple$_{s2}$. Both SJimple$_{s1}$ and SJimple$_{s2}$ are obtained from SJimple$_{Prog}$.

– Commit the edits and update the dataflows. Committing the translated edits on SJimple$_{Prog}$ can produce Jimple$_{Prog'}$, which is equivalent to SJimple$_{Prog'}$ with respect to the points-to relations. Since the temporary variables used in the changed instructions may conflict with those in SJimple$_{Prog}$, we update them in Jimple$_{Prog'}$. Specifically, any temporary variable introduced into an updated segment needs to be assigned with a new name such that it does not conflict with the variables outside the segment.

– Mark up the change information in Jimple$_{Prog'}$. The change information can be summarized by comparing Jimple$_{Prog}$ and Jimple$_{Prog'}$: an instruction is marked with a label "+" if it exists in Jimple$_{Prog'}$ but not in Jimple$_{Prog}$, or marked with "–" on the contrary.

For example, Fig. 3 shows the transformation of the edits on BubbleSort to those on its Jimple's code. The first edit is transformed into three insertion edits, each of which inserts an instruction into the constructor (i.e., BubbleSort()). The second edit is transformed into ten insertion edits which insert instructions into the method sort(boolean). Here ki represents an identification number of an inserted instruction, $t$i$ represents a temporary variable, and _ represents an intermediate method resulting from the commitment of the preceding edit. Notice that the resulting Jimple's code is incomplete in that the control flows are omitted, while it is sufficient to use the code to produce the inclusion constraints and solve them.

3.4 Construction of Inc-PAG

Our analysis computes the annotated points-to graph called *Inc-PAG*. A PAG describes the points-to relations in the program being analyzed. We extend PAG to *Inc-PAG* in that (1) each edge or vertex holds a flag stating whether it has been changed or not; and (2) the label list on each edge records the changes of the lines of Jimple's code. An Inc-PAG is defined as a labelled directed graph $< V, E >$ where

- V contains a set of vertices representing the memory locations;
- E contains a set of edges representing the points-to relations;
- Each vertex is associated with a points-to set;
- Each edge is attached with a list of labels denoting the lines of Jimple's code that may contribute to the edge and the change information. For example, $v_1 {\rightarrow}_{0,-1,+2} v_2$ indicates that the lines of code for $v_1 \rightarrow v_2$ are changed from $\{0, 1\}$ to $\{0, 2\}$;
- Each vertex or edge is assigned with a flag "+" (or "–") if it is newly inserted into (or deleted from) the graph.

Let `Prog` be changed to `Prog'`. Let $PAG_{\texttt{Prog}}$ be the pointer assignment graph of `Prog`. *Inc-*$PAG_{\texttt{Prog}'}$ is constructed by propagating the edits in the editscript to $PAG_{\texttt{Prog}}$, as next shows:

1. *Inc-*$PAG_{\texttt{Prog}'} = PAG_{\texttt{Prog}}$;
2. Iterating through $\texttt{Jimple}_{\texttt{Prog}'}$ with change information and updating *Inc-*$PAG_{\texttt{Prog}'}$.
   (a) For a Jimple's instruction L having a label "+", create an edge (say `a`→ `b`) representing the points-to relation introduced by L, and either add "$+L$" to the label list of `a`→`b` if *Inc-*$PAG_{\texttt{Prog}'}$ contains it, or add $a \rightarrow_{+L} b$ into *Inc-*$PAG_{\texttt{Prog}'}$ otherwise.
   (b) For an instruction L having a label "–", identify its edge in *Inc-*$PAG_{\texttt{Prog}'}$, and change the label "L" to "$-L$".
3. Assign a flag "+" (or "–") to an edge or a vertex if it is newly inserted into (or deleted from) the points-to graph. Specifically, a vertex is assigned with "–" if it is not connected by the others in the graph, and an edge is assigned with "–" if its label list only contains the labels with the prefix "–".

Figure 4 shows a simplified Inc-PAG of `BubbleSort'`, where an abstract representation of the PAG of `BubbleSort` is shown in the top part, the bottom part contains a set of vertices and edges introduced by the Jimple's instructions shown in Fig. 3, the vertices (e.g., `L13` and `L25`) represent the heap references

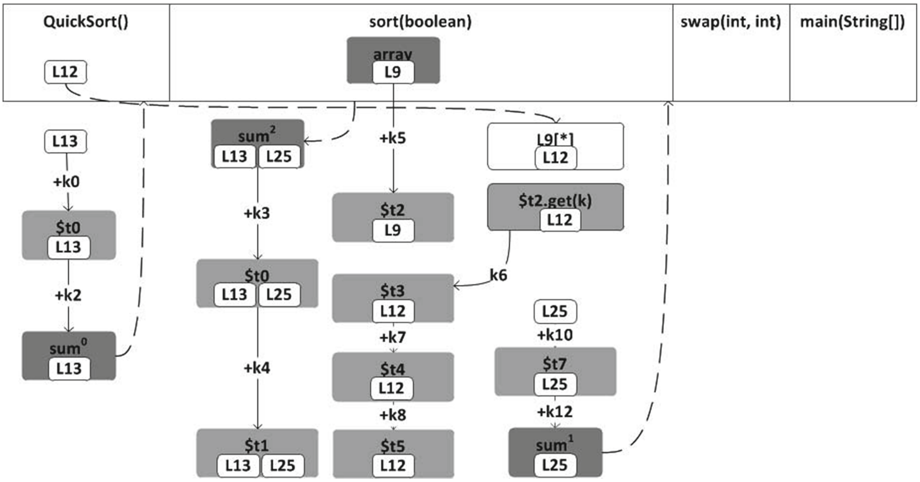


Fig. 4. An Inc-PAG of `Prog'`.

at lines 13 and 25, respectively, the vertices (e.g., `array`, `sum`, and $t0$) represent the variables used in the program, and the other vertices represent the filed references and the concrete fields, respectively. Note that in Fig. 4, (1) each edge in the incremental part is assigned with a label denoting the inserted instruction; (2) since all of the edges and vertices in the incremental part are newly inserted, we omit their flags ("+") in the figure; and (3) some dashes are used to represent the edges between a vertex in the incremental part and some vertices in the original PAG.

When the Inc-PAG of `BubbleSort'` is constructed, an insertion edit on the Jimple's instruction is translated to an insertion edit on the graph. For example, we have

```
   Insert(BubbleSort(),"+k0: $t0 = new java.lang.
Integer;")⇒
   Insert(Inc-PAG_BubbleSort', k0: L13→$t0)
```

3.5 Solving of Points-to Sets

Given $PAG_{\texttt{Prog}}$ with solved points-to sets, *Inc-PAG*$_{\texttt{Prog}'}$ can be re-solved by patching up the points-to sets in $PAG_{\texttt{Prog}}$. Furthermore, any patch to *Inc-PAG*$_{\texttt{Prog}'}$ (increment or decrement) needs to be propagated throughout the graph. The process of re-solving of inclusion constraints is given as follows:

- Traverse *Inc-PAG*$_{\texttt{Prog}'}$ and initialize the points-to sets in *Inc-PAG*$_{\texttt{Prog}'}$. For each vertex v in *Inc-PAG*$_{\texttt{Prog}'}$, we let

$$pt(v) = \begin{cases} pt(matching(v)) & \text{when } v \text{ is matched with a vertex in } PAG_{\texttt{Prog}}; \\ \varnothing & \text{when } v \text{ has a flag "+".} \end{cases}$$

Here *matching*(v) returns the vertex in $PAG_{\texttt{Prog}}$ matched with v;

- For each edge $v_1 \rightarrow v_2$ having a flag "+", propagate an increment of the points-to set of v_1 to that of v_2. Let $e \in pt(v_1)$. The propagation adds e^+ to the points-to set of v_2 if $e \notin pt(v_2)$ (or $e^+ \notin pt(v_2)$).
- For each edge $v_1 \rightarrow v_2$ having a flag "–", propagate a decrement of the points-to set of v_1 to that of v_2. Let $e \in pt(v_1)$. The propagation adds e^- to the points-to set of v_2.
- Iterate the next two steps until the points-to sets in $\textit{Inc-PAG}_{\texttt{Prog'}}$ are unchanged, i.e., a fixed point is reached.
  1. For each vertex, if its points-to set is incremented or decremented, propagate the increments or the decrements along all of its out-edges.
     - Let $v_1 \rightarrow v_2$ and $e^+ \in pt(v_1)$. The propagation adds e^+ to the points-to set of v_2 if $e \notin pt(v_2)$.
     - Let $v_1 \rightarrow v_2$ and $e^- \in pt(v_1)$. The propagation adds e^- to the points-to set of v_2.
  2. Commit the increments and decrements on the points-to sets. Specifically, a decrement e^- is committed on the points-to set of v if e cannot reach v, i.e., none of the sources of the in-edges can point to e.

Figure 4 also shows the propagation of the points-to sets throughout the graph. After propagation, $\texttt{sum}^0$ points to the heap reference(s) at line 13, $\texttt{sum}^1$ points to that (those) at line 25, and $\texttt{sum}^2$ can either point to the heap reference(s) created at line 13 or that (those) at line 25.

We can also take the above activity as a process of translating the edits in $\texttt{EditScript}_{\texttt{BubbleSort}}$ into the edits on the points-to sets, as next shows. The first edit is translated into adding $\{\texttt{sum}^0 \rightarrow \texttt{L13}\}$ into the points-to sets of the program. The second edit is translated into adding $\{\texttt{sum}^1 \rightarrow \texttt{L13}, \texttt{sum}^2 \rightarrow \texttt{L13}, \texttt{sum}^2 \rightarrow \texttt{L25}\}$ into the points-to sets.

1. `Insert(BubbleSort(),"13: sum`^{0}`=new Integer(0);")`$\Rightarrow$
   `Points-to-Set = Points-to-Set`$\cup\{\texttt{sum}^0 \rightarrow \texttt{L13}\}$
2. `Insert(sort(boolean),"25: sum`^{1}`=new Integer(sum`^{2}`+ array.`
   `get(k));")`$\Rightarrow$
   `Points-to-Set = Points-to-Set`$\cup\{\texttt{sum}^1 \rightarrow \texttt{L13}, \texttt{sum}^2 \rightarrow \texttt{L13}, \texttt{sum}^2 \rightarrow \texttt{L25}\}$

3.6 Discussion

An engineer who wants to use Inc-PTA may concern about the correctness of the approach and its complexity. By saying that the approach is correct we mean that the final result computed by using Inc-PTA needs to be same as that computed by using any traditional points-to analysis algorithm. Let $PAG_{\texttt{Prog'}}$ be the pointer assignment graph of `Prog'` that is obtained in the Andersen-style points-to analysis. The correctness of Inc-PTA is guaranteed by two respects: (1) $\textit{Inc-PAG}_{\texttt{Prog'}}$ is isomorphic with $PAG_{\texttt{Prog'}}$, which means that the incremental analysis process does not introduce any inclusion constraints besides those in $PAG_{\texttt{Prog'}}$, except that we may assign different names to the temporary variables;

(2) Inc-PTA solves the constraints in an incremental or decremental manner, which does not alter the natural intrinsics of constraint solving in that the final result meets all of the constraints.

Inc-PTA is an approach to incremental points-to analysis, which reduces the cost of analysis in that we do not need to rebuild the PAG for the program from scratch and re-solve all of the constraints but only patch up them in accordance with the edits on the program. Although the costs that can be reduced are different case by case, an investigation indicates that Inc-PTA provides with a lightweight solution to points-to analysis. In our case study (i.e., the `BubbleSort` program), only about 10.4 percent of the Jimple's code are changed, which implies that a similar proportion of the PAG is edited and of the constraints are re-solved. However, an experiment still needs to be conducted to evaluate the correctness and complexity of Inc-PTA, which will be remained as one of our future work.

4 Related Work

Efforts have been conducted to improve the precision and efficiency of points-to analysis. Flow Sensitive Points-to Analysis (FSPA) can help achieve precise points-to relations by distinguishing the variables at different program points, while the analysis suffers a shortcoming of its low efficiency when used to analyze large-scale software systems or software with libraries [6]. Hardekopf and Lin have introduced the lazy and the eager techniques for inclusion-based pointer analysis which improve the efficiency of the analysis without reducing precision [16]. They have also adopted partial SSA form to increase the efficiency of FSPA [17,18], and used a less precise auxiliary pointer analysis to create the def-use chains, which enables the sparsity of FSPA [8]. Gulwani, Srivastava, and Venkatesan performs points-to analysis by solving of constraints on the points-to relations. In an analysis run a program is translated into constraints that are solved using some off-the-shelf constraint solvers to yield desired result [19]. Visser and Dwyer have developed a Green solver which reduces constraints to a simple form, allowing for reusing constraint solutions either within an analysis run or in other runs [20].

Meanwhile researchers have been exploring the algorithms for incremental data flow analysis for over three decades. ACINCF and ACINCB were the early incremental update algorithms for forward and backward data flow analysis, respectively [21]. Pollock and Soffa have presented a technique to incrementally update solutions to both union and intersection data flow problems [22]. Specially, for such problems, some program changes are incrementally incorporated into the data flow sets, while others are handled by a two phase strategy: the first phase updates the data flow sets to overestimate the effects of the program changes, and the second one incrementally updates the affected data flow sets to reflect the changes. Burke and Ryder have presented a model of data flow analysis and fixed point iteration solution procedures, and then summarized some incremental data flow analysis techniques [23]. Carroll and Polychronopoulos have presented an algorithm for incremental inter-procedural data flow analysis, where the strongly connected component (SCC) ordering is used to determine which functions need to be re-evaluated [24,25].

Some incremental points-to analysis algorithms have been developed in the last decade. Yur, Ryder, and Landi have proposed an approach to incremental approximation of a flow- and context-sensitive alias analysis, which falsifies the aliases affected by the changes [26]. Vivien and Rinard have observed that the incremental analysis of a small region of a program can provide with the benefit of a whole-program analysis, and then presented a pointer and escape analysis that incrementally analyzes only some parts of the program for delivering the results [27]. Saha and Ramakrishnan [28] have devloped practical implementations of incremental program analyzers. Kodumal and Aiken [29] have developed the Banshee toolkit, which allows constraint systems to be rolled back to some previous state for a code change and re-analyze the program from that state. Lu and Xue have taken the incremental points-to analysis with CFL reachability [9]. By tracing some CFL-reachable paths, the engineers can precisely identify and recompute on demand the points-to sets affected by the program changes.

Compared with the above related work, Inc-PTA provides with an approach to incremental points-to analysis, but takes a trade-off between the efficiency and the easiness of implementation. The principle of Inc-PTA is consistent with the traditional iterative algorithm, and thus it is more intuitive for the engineers to implement the algorithm. Inc-PTA also does not require to identify the program paths and the points-to relations related to the program changes. Thus it refrains from adopting some slicing or path-tracing techniques and further refrains from the precision loss caused by some conserved algorithms.

5 Conclusions and Future Work

Inclusion-based points-to analysis is extremely inefficient in both the time and space, especially when the objective programs are frequently changed. In this paper we have proposed an approach Inc-PTA to incremental points-to analysis. By using Inc-PTA, an engineer solves the points-to constraints on a program in an edit propagation style, which makes the analysis adapt to the program changes and as well reduce the cost of re-generating the constraints when the program is frequently changed.

A potential extension to Inc-PTA is to paralleling the points-to analysis approach. It is noteworthy that some constraints may be independent from the others and can be solved individually, which makes it believed that the principle of Inc-PTA can be applied, with slight modifications, in support of paralleling the points-to analysis. In addition, we would adopt some heuristic algorithms to solve the constraints such that the points-to sets may be quickly guessed and used in the analysis process in order to reduce the total number of iterations.

Acknowledgments. We would like to thank the anonymous reviewers for their valuable and thorough comments. This work is supported by the National Natural Science Foundation of China (Grant No. 91118004 and 61100051).

References

1. Steensgaard, B.: Points-to analysis in almost linear time. In: Boehm Jr., H.J., Shackle, G.L.S. (eds.) POPL, pp. 32–41. ACM Press, New York (1996)
2. Andersen, L.O.: Program analysis and specialization for the c programming language. Master's thesis, University of Copenhagen (1994)
3. Méndez-Lojo, M., Mathew, A., Pingali, K.: Parallel inclusion-based points-to analysis. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) OOPSLA, pp. 428–443. ACM, New York (2010)
4. Méndez-Lojo, M., Burtscher, M., Pingali, K.: A gpu implementation of inclusion-based points-to analysis. In: Ramanujam, J., Sadayappan, P. (eds.) PPOPP, p. 107. ACM, New York (2012)
5. Lhoták, O., Smaragdakis, Y., Sridharan, M.: Pointer analysis (dagstuhl seminar 13162). Dagstuhl Rep. **3**(4), 91–113 (2013)
6. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural dataflow analysis in the presence of large libraries. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 2–16. Springer, Heidelberg (2006)
7. Chase, D.R., Wegman, M.N., Zadeck, F.K.: Analysis of pointers and structures. In: Fischer, B.N. (ed.) PLDI, pp. 296–310. ACM, New York (1990)
8. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: IEEE CGO, pp. 289–298 (2011)
9. Lu, Y., Shang, L., Xie, X., Xue, J.: An incremental points-to analysis with CFL-reachability. In: Jhala, R., De Bosschere, K. (eds.) Compiler Construction. LNCS, vol. 7791, pp. 61–81. Springer, Heidelberg (2013)
10. Jenista, J., Eom, Y., Demsky, B.: Using disjoint reachability for parallelization. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 198–224. Springer, Heidelberg (2011)
11. Vallee-Rai, R., Hendren, L.J.: Jimple: Simplifying java bytecode for analyses and transformations (1998)
12. Lhoták, O., Hendren, L.: Scaling java points-to analysis using SPARK. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 153–169. Springer, Heidelberg (2003)
13. Gall, H., Fluri, B., Pinzger, M.: Change analysis with evolizer and changedistiller. IEEE Softw. **26**(1), 26–33 (2009)
14. Fluri, B., Würsch, M., Pinzger, M., Gall, H.: Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. **33**(11), 725–743 (2007)
15. Vallée-Rai, R.C.P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: CASCON. In: MacKay, S.A., Johnson, J.H. (eds.) Soot-A Java Bytecode Optimization Framework, p. 13. IBM, San Diego (1999)
16. Hardekopf, B., Lin, C.: The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 290–299. ACM, New York (2007)
17. Tok, T.B., Guyer, S.Z., Lin, C.: Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 17–31. Springer, Heidelberg (2006)
18. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 226–238. ACM, New York (2009)
19. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI, pp. 281–292. ACM, New York (2008)

20. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: reducing, reusing and recycling constraints in program analysis. In: Tracz, W., Robillard, M.P., Bultan, T. (eds.) SIGSOFT FSE, p. 58. ACM, New York (2012)
21. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.D.: Conversion of control dependence to data dependence. In: Wright, J.R., Landweber, L., Demers, A.J., Teitelbaum, T. (eds.) POPL, pp. 177–189. ACM, New York (1983)
22. Pollock, L.L., Soffa, M.L.: An incremental version of iterative data flow analysis. IEEE Trans. Soft. Eng. **15**(12), 1537–1549 (1989)
23. Burke, M.G., Ryder, B.G.: A critical analysis of incremental iterative data flow analysis algorithms. IEEE Trans. Soft. Eng. **16**(7), 723–728 (1990)
24. Carroll, S., Polychronopoulos, C.D.: A framework for incremental extensible compiler construction. Int. J. Parallel Program. **32**(4), 289–316 (2004)
25. Carroll, S., Polychronopoulos, C.D.: A framework for incremental extensible compiler construction. In: Banerjee, U., Gallivan, K., González, A. (eds.) ICS, pp. 53–62. ACM, New York (2003)
26. Yur, J.S., Ryder, B.G., Landi, W.: An incremental flow- and context-sensitive pointer aliasing analysis. In: Boehm, B.W., Garlan, D., Kramer, J. (eds.) ICSE, pp. 442–451. ACM, New York (1999)
27. Vivien, F., Rinard, M.C.: Incrementalized pointer and escape analysis. In: Burke, M., Soffa, M.L. (eds.) PLDI, pp. 35–46. ACM, New York (2001)
28. Saha, D., Ramakrishnan, C.R.: Incremental and demand-driven points-to analysis using logic programming. In: Barahona, P., Felty, A.P. (eds.) PPDP, pp. 117–128. ACM, New York (2005)
29. Kodumal, J., Aiken, A.: Banshee: a scalable constraint-based analysis toolkit. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 218–234. Springer, Heidelberg (2005)

A Memory Management Mechanism for MSVL

Kai Yang, Zhenhua Duan(✉), and Cong Tian

ICTT and ISN Lab, Xidian University, Xi'an 710071, China
kaiyang@stu.xidian.edu.cn, zhenhua_duan@126.com, ctian@mail.xidian.edu.cn

Abstract. This paper presents a memory management mechanism for programs of Modeling, Simulation and Verification Language (MSVL) which is a subset of Projection Temporal Logic (PTL) with framing technique. Framing operator is defined in MSVL and is concerned with the persistence of the values of variables from one state to another. Based on framing technique, we implement a memory management mechanism for MSVL programs. In short, memory can be allocated and released dynamically according to the framing operator. As a result, the efficiency can be improved and the memory can be used more effectively when MSVL programs are executed in MSV which is a toolkit developed for the purpose of modeling, simulation and verification of MSVL programs.

Keywords: Temporal logic · MSVL · Framing technique · Memory management

1 Introduction

Modeling, Simulation and Verification Language (MSVL) is a subset of Projection Temporal Logic (PTL) with framing technique [5,6,9]. It can be used to simulate, model and verify software and hardware systems [4,7]. A toolkit named MSV has been developed in C++ for the purpose of modeling, simulation and verification of MSVL programs. Further, MSVL programs are executed by interpreting in MSV. Symbol table is a key module in MSV, which is employed to simulate the stack and heap in a compiler, namely, saving information of variables during the execution of an MSVL program. What we care about is how to keep the size of the symbol table as small as possible to save memory.

Memory management is a technique caring about the memory of a computer when a system is running. It focuses on how to allocate and release memory resource efficiently [10,15,16]. In a program, the compiler or interpreter must be told explicitly or implicitly when a memory cell should be allocated or released. A good memory management mechanism is vital for a practical programming language.

In a conventional programming language such as C or JAVA [2,11–13], as we all know, the system will allocate a memory cell for a dynamic variable

This research is supported by the NSFC Grant No. 61133001, 61272117, 61322202, 61420106004, and 91418201.

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2014, LNCS 8979, pp. 179–188, 2015.
DOI: 10.1007/978-3-319-17404-4_12

when it is declared and maintain the cell within the scope of the variable. If a variable has not been assigned a new value, the current value of the variable is kept. However, the situation is different in a temporal logic programming language such as MSVL. A temporal logic program is executed over a sequence of states and the values of variables do not inherit their old values automatically. Therefore, a framing technique is introduced to deal with this problem. Framing is concerned with how the value of a variable from one state can be carried to the next one.

Introduction of framing technique to temporal logic programming is motivated by both practical and theoretical aspects: improving the efficiency of a program and synchronizing communication for parallel processes [3,6,17]. A variable in a MSVL program can be framed, non-framed or framed over a subinterval. Intuitively, if a variable is framed, it always keeps its old value over an interval if no assignment to it is encountered. Otherwise, the value of the variable will be *nil* (not defined) if no assignment to it is encountered. Obviously, if a variable is non-framed, its value in the history will not be referenced and MSV needs not maintain a memory cell for it to save memory.

In our algorithm, in brief, MSV allocates and releases memory according to variables' scopes similar to other programming languages. To this end, the scopes of framed dynamic variables and non-framed dynamic variables are given respectively in our method. Obviously, the scope of a static variable is the whole interval, hence, static variables will be kept in the symbol table from the beginning to the end of the execution. MSV allocates memory unit for a dynamic variable (adding the variable to the symbol table) framed or non-framed at the beginning of the its scope and releases the memory unit (removing the variable from the symbol table) of it when its scope ends. Through our method, MSV will release the memory of a useless variable timely and as a consequent the memory will be saved.

The remainder of this paper is organized as follows. In Sect. 2, the framing technique in temporal logic programming is briefly introduced. The syntax and semantics of the MSVL and normal form of a MSVL program are given in Sect. 3. Section 4 presents a memory management algorithm for MSVL programs. A case study is given in Sect. 5 and conclusion is drawn in Sect. 6.

2 Framing Techniques

Framing is concerned with whether or not the value of a variable should be persisted over an interval. Intuitively, if a variable x is framed in an interval, the value of x in the previous will be inherited over an interval if there are no new assignments to x are encountered. There are state framing (lbf) and interval framing ($frame$) operators. Specifically, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state. A variable is framed over an interval if it is framed at every state over the interval. The following are the definitions of lbf and $frame$.

$$\begin{aligned} lbf(x) &\overset{\text{def}}{=} \neg p_x \rightarrow \exists b : (\odot x = b \wedge x = b) \\ frame(x) &\overset{\text{def}}{=} \Box(more \rightarrow \bigcirc lbf(x)) \\ frame(x_1, ..., x_n) &\overset{\text{def}}{=} frame(x_1) \wedge ... \wedge frame(x_n) \end{aligned}$$

where b is a static variable and p_x an atomic proposition associated with state (dynamic) variable x. p_x holds if x is assigned a new value at the current state, otherwise $\neg p_x$ holds. In addition, p_x cannot be used for other purpose. Here, $\neg$, $\wedge$ and $\rightarrow$ are defined as usual. $\odot x$ denotes the value of x at the previous state. For an MSVL program p, $\Box p$ means that p holds at every state over the whole interval and $\bigcirc p$ indicates that p holds at the next state. In addition, *more* means that the current state is not the final state of the interval.

3 Modeling, Simulation and Verification Language

The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$\begin{aligned} e &::= \ n \mid x \mid \ \bigcirc x \mid \odot x \mid e_0 \ op \ e_1 \ (op ::= + | - \mid * \mid \backslash \ |mod) \\ b &::= \ true \mid fasle \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \ \wedge \ b_1 \end{aligned}$$

where n is an integer and x a variable. Some elementary statements in MSVL that are used in this paper are presented as follows. Please refer to [7] for the definition of all statements in MSVL.

$Termination : empty$ $\qquad Assignment : x = e$
$P - I - Assignment : x \Leftarrow e$ $\qquad Conjunction : p \wedge q$
$Selection : p \vee q$ $\qquad Next : \bigcirc p$
$Always : \Box p$ $\qquad Sequence : p; q$

$$Conditional : if \ b \ then \ p \ else \ q \overset{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$$
$$While : while \ b \ do \ p \overset{\text{def}}{=} (p \wedge b)^* \wedge \Box(empty \rightarrow \neg b)$$

where x denotes a variable, e an arbitrary arithmetic expression, b a boolean expression, and p and q programs of MSVL.

The termination statement *empty* means that the current state is the final state of the interval over which the program is executed. The assignment $x = e$ denotes 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*. The conjunction statement $p \wedge q$ denotes that p and q are executed in a concurrent manner and share all the variables during the execution, and p and q have the same interval lengths. $p \vee q$ means p or q is selected randomly to execute. The next statement $\bigcirc p$ means that p holds at the next state. Intuitively, the sequence statement $p; q$ means that program p is executed from the current state until its termination, then program q is executed. The conditional statement $if \ b \ then \ p \ else \ q$, as in the conventional programming language, means that if the condition b is evaluated *true* then process p is executed, otherwise process q is executed. Statement *while* b *do* p is the loop statement which denotes that process p will be repeatedly executed until condition b becomes *false*.

3.1 Normal Form of MSVL Programs

Definition 1 (Normal Form of MSVL Programs). *An MSVL program* q *is in its Normal Form if* q *has been rewritten in the following form:*

$$q \overset{\text{def}}{=} \bigvee_{i=1}^{k} q_{ei} \wedge \mathsf{empty} \vee \bigvee_{j=1}^{h} q_{cj} \wedge \bigcirc q_{fj}$$

where $k, h \geq 0$ $(k + h \geq 1)$*, and*

- q_{fj} *is a general MSVL program;*
- *each* q_{ei} *and* q_{cj} *is either* true *or a state formula of the form* $(x_1 = e_1) \wedge \ldots \wedge (x_l = e_l) \wedge \dot{p}_{x_1} \wedge \ldots \wedge \dot{p}_{x_m}$ *where* $e_i \in D\,(1 \leq i \leq l)$*, and* $\dot{p}_x$ *denotes* p_x *or* $\neg p_x$*,* $l \geq 0$*,* $m \geq 0$*, and* $l + m \geq 1$*.*

Specially, we call q_{cj} the present component, q_{fj} the future component, and q_{ei} the termination component in a normal form.

The following theorem has been proved in [7].

Theorem 1. *Any MSVL program* p *can be rewritten into its normal form.*

Theorem 1 tells us that for any MSVL program p there is a program q in the normal form such that $p \equiv q$.

4 Memory Management for MSVL

As known to all, every variable has its scope in a conventional program. For instance, in a C program, for an dynamic variable declared at the beginning of a block, the scope is the block in which the name is declared. The compiler or interpreter allocates and releases memory according to the scope of a variable. When a dynamic variable is declared, the compiler will allocate a memory unit for it in the variable stack for it and maintain the memory unit until the end of the variable's scope. Throughout it, the value of the variable may be modified by a process. In other words, the current value of the variable remains until a new assignment to it is encountered.

In order to implement memory management for MSVL programs, the scopes of variables (all variables are dynamic variables if not specified specially) in MSVL programs also need to be specified. Because the value of a framed variable will be kept in the framed interval while the value of a non-framed variable will not be taken to the next state. Informally, the scope of a framed variable is the framed interval associated with it and the scope of a non-framed variable is the state in which it is assigned.

Although there are difference among scopes, memory management for framed and non-framed variables can be dealt with in the same way with our method. Formally, the state memory management algorithm and interval memory management algorithm for MSVL programs are given as StateMM and IntervalMM. As their names show, StateMM deals with the memory allocation and release at a state while IntervalMM cares about the memory allocation and release over a whole interval. Further, considering efficiency, a symbol table in MSV is implemented with the data structure map in STL of C++ [14].

Algorithm 1. StateMM(q, ST)

```
Input:
    $q$ is a disjunct in $NF(P)$ where $P$ is the program to be executed at the current
    state ST is the symbol table
Output:
    ST
if !ST.empty() then
    for each variable $x$ in ST do
        if $p_x$ is not in the present component of $q$ then
            if $\neg p_x$ is not in the present component of $q$ then
                ST.erase($x$);
            end if
        end if
    end for
end if
for each state formula $x = e$ in the present component of $q$ do
    if !ST.find(x) then
        ST.insert(x);
        Assign(x, e);
    else
        Assign(x, e);
    end if
end for
```

In Algorithm StateMM, NF is an algorithm defined in [8] for translating a preprocessed MSVL to its normal form. $Assign(x, e)$ assigns value e to variable x.

To execute an MSVL program, at the beginning of each state, MSV translates the program to its normal form and then one of the disjuncts in the normal form will be selected randomly to execute. Before the present component of the selected disjunct is executed, MSV checks the variables whose life cycles have ended at this state and the memory space of these variables will be released. Specifically, for each variable x in the symbol table (if the symbol table is not empty), MSV checks if there is p_x or $\neg p_x$ in the present component firstly. If no, MSV removes variable x from the symbol table. Then the assignment statements in present component will be executed. For instance, when $x = 3$ is to be executed, MSV will add variable x to the symbol table and then assign 3 to x if the variable x is not in the symbol table.

Algorithm 2. IntervalMM(P, ST)

```
Input:
    $P$ is the program to be executed
    ST is the symbol table
Output:
    ST
ST=$\emptyset$;
if existing a next state in $P$ and the program to be executed at next state is $q$ then
    NF(q);
    StateMM(q, ST);
else
    ST.clear();
end if
```

It is known from the definition of $frame(x)$ that for a framed variable x, starting from the second state of the framed interval, there must be p_x in the present component if x is assigned a new value at current state, or $\neg p_x$ in the present component otherwise. As a consequence, in our method, the memory cell of variable x will be maintained until the interval of $frame(x)$ is terminated. It is trivially correct for non-framed variables. For a non-framed variable x, if x is not assigned at the current state, p_x and $\neg p_x$ will not appear in the present component and the memory space of variable x will be released in this case.

The following is a MSVL program called GCD which computes the greatest common divisor of x and y with the result saved in variable g. The program is an implementation of Euclid's well known algorithm [1].

$$\begin{array}{l}
frame(x,\, y)\, and\, (\\
\quad x <== 6 \;\; and \;\; y <== 4 \; and \; empty; \\
\quad while(x\,! = \; y)\, \{ \\
\qquad if(x \; > \; y) \;\; then \;\; \{\, x := x - y\} \\
\qquad else \;\; \{y := y - x\} \\
\quad \}; \\
\quad g := x \\
)
\end{array}$$

In the above program, x and y are framed dynamic variables while g is a non-framed dynamic variable. Variable g is not assigned until the last state (state 3). Hence, its value is unspecified at state 0, 1 and 2. The computation which gives the values of the variables of each state in the program is depicted in Fig. 1. The mark (?) means that the value of the variable is unknown at the state.

According to algorithm IntervalMM, the symbol table is empty initially. Firstly, we translate the program to its normal form and obtain the present and future component of each state. Only the present components are given here because we do not care about the future components in our algorithm. p_c^0 stands for the present component of state 0 and so on. p_e^3 denotes the termination component of state 3. The left hand side of Fig. 2 shows the symbol table

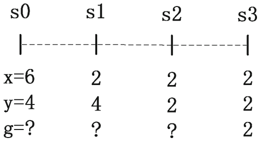


Fig. 1. Computation of GCD

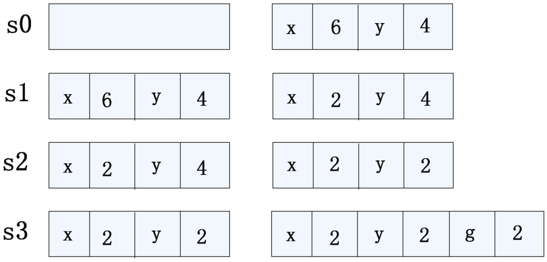


Fig. 2. Symbol tables of GCD

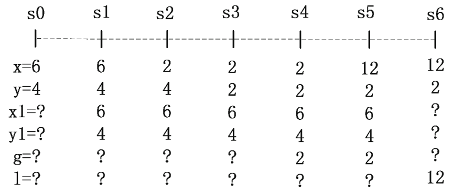
before the present component of state i is executed and the right hand side of Fig. 2 shows the symbol table after the present component of state i is executed. The types of the variables are ignored here for clarity (all variables are integers actually).

$$
\begin{aligned}
p_c^0 &\equiv x = 6 \land p_x \land y = 4 \land p_y \\
p_c^1 &\equiv x = 2 \land p_x \land y = 4 \land \neg p_y \\
p_c^2 &\equiv x = 2 \land \neg p_x \land y = 2 \land p_y \\
p_e^3 &\equiv x = 2 \land \neg p_x \land y = 2 \land \neg p_y \land g = 2 \land p_g
\end{aligned}
$$

Furthermore, frame statements that are sequential and nested in a MSVL program can be dealt with in our method. The latter is more complex and we use the following example called LCM to illustrate this case. Firstly, the program computes the greatest common divisor of x and y with the result eventually saved in g. Then g is used to compute the least common multiple of x and y. The eventual result is saved in variable l.

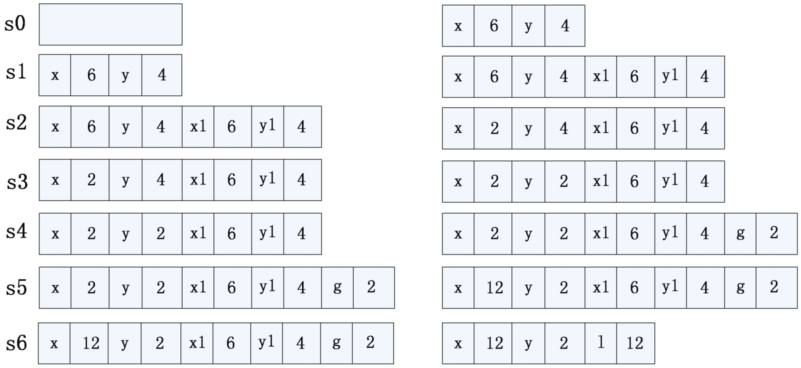
```
$frame(x, y)\ and\ ($
  $x <== 6\ \ and\ \ y <== 4\ and\ empty;$
  $frame(x1, y1)\ and\ ($
    $x1 := x\ \ and\ \ y1 := y;$
    $while(x\,! =\ y)\ \{$
        $if(x\ >\ y)\ then\ \ \{x := x - y\}$
        $else\ \ \{y := y - x\}\};$
    $frame(g)\ and\ ($
        $g := x;$
        $x := (x1 * y1)/g));$
  $l := x$
$)$
```

The computation and symbol table of each state are given in Figs. 3 and 4 respectively. In the program, variables x and y are framed at the overall interval while variables $x1$ and $y1$ are framed at a subinterval of the interval $frame(x, y)$ is executed, and variable g is framed at a subinterval of the interval $frame(x1, y1)$ is executed. Initially, the symbol table is empty at state 0. Variables x and y are added to the symbol table at state 0. Similarly, variables $x1$ and $y1$ are added to the symbol table at state 1, and variable g is added to the symbol table at state 4. Considering their scopes, variables x and y are kept in the symbol table over the whole interval. Variables $x1$, $y1$, and g are removed from the symbol table at state 6 because the intervals associated with $frame(x1, y1)$ and $frame(g)$ are terminated at this state.



s0	s1	s2	s3	s4	s5	s6
x=6	6	2	2	2	12	12
y=4	4	4	2	2	2	2
x1=?	6	6	6	6	6	?
y1=?	4	4	4	4	4	?
g=?	?	?	?	2	2	?
1=?	?	?	?	?	?	12

Fig. 3. Computation of LCM

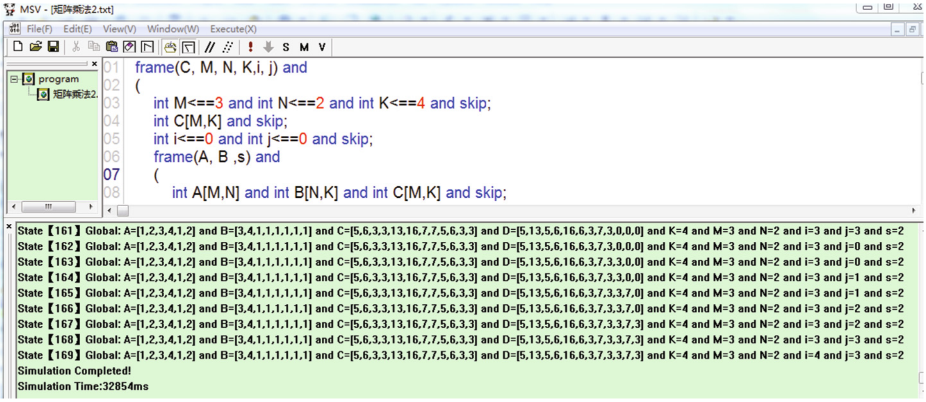


s0											x	6	y	4						
s1	x	6	y	4							x	6	y	4	x1	6	y1	4		
s2	x	6	y	4	x1	6	y1	4			x	2	y	4	x1	6	y1	4		
s3	x	2	y	4	x1	6	y1	4			x	2	y	2	x1	6	y1	4		
s4	x	2	y	2	x1	6	y1	4			x	2	y	2	x1	6	y1	4	g	2
s5	x	2	y	2	x1	6	y1	4	g	2	x	12	y	2	x1	6	y1	4	g	2
s6	x	12	y	2	x1	6	y1	4	g	2	x	12	y	2	1	12				

Fig. 4. Symbol tables of LCM

5 A Case Study

Matrix operation is important in digital image processing. Using a matrix to store a large image always takes up a lot of memory space. We implement an algorithm for calculating the product matrix C of two matrices A and B and the transpose matrix D of the product matrix C in MSVL. Part of the execution result in MSV is shown in Fig. 5. In the program, four two-dimensional arrays A, B, C and D are used to denote the matrices A, B, C and D respectively. Variable C is framed over the whole interval while A and B are framed in a



```
frame(C, M, N, K,i, j) and
(
    int M<==3 and int N<==2 and int K<==4 and skip;
    int C[M,K] and skip;
    int i<==0 and int j<==0 and skip;
    frame(A, B ,s) and
    (
        int A[M,N] and int B[N,K] and int C[M,K] and skip;
```

```
State【161】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,0,0,0] and K=4 and M=3 and N=2 and i=3 and j=3 and s=2
State【162】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,0,0,0] and K=4 and M=3 and N=2 and i=3 and j=0 and s=2
State【163】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,3,0,0] and K=4 and M=3 and N=2 and i=3 and j=0 and s=2
State【164】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,3,0,0] and K=4 and M=3 and N=2 and i=3 and j=1 and s=2
State【165】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,3,7,0] and K=4 and M=3 and N=2 and i=3 and j=1 and s=2
State【166】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,3,7,0] and K=4 and M=3 and N=2 and i=3 and j=2 and s=2
State【167】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,3,7,3] and K=4 and M=3 and N=2 and i=3 and j=2 and s=2
State【168】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,3,7,3] and K=4 and M=3 and N=2 and i=3 and j=3 and s=2
State【169】Global: A=[1,2,3,4,1,2] and B=[3,4,1,1,1,1,1,1] and C=[5,6,3,3,13,16,7,7,5,6,3,3] and D=[5,13,5,6,16,6,3,7,3,3,7,3] and K=4 and M=3 and N=2 and i=4 and j=3 and s=2
Simulation Completed!
Simulation Time:32854ms
```

Fig. 5. Matrix operation

subinterval of the interval where $frame(C)$ is executed, D is framed at another subinterval of the interval where $frame(C)$ is executed. In the algorithm, after getting the product matrix of A and B, matrices A and B will be useless and we do not need to maintain them in the memory of for saving memory. Furthermore, the larger the size of A or B the more the memory will be saved. This is of great importance for processing large matrix in practice. Table 1 shows the memory that the symbol tables takes (in the old version of MSV and the new version of MSV) and the memory saved for different sizes of matrices.

Table 1. Comparison of memory usage

Size		Old (Mb)	New (Mb)	Memory saved (Mb)
A	B			
300*200	200*400	1.52	1.04	0.48
600*400	400*800	6.08	4.16	1.92
900*600	600*1200	13.68	9.36	4.32
1200*800	800*1600	24.32	16.64	7.68

6 Conclusion

This paper presents a memory management mechanism for MSVL programs based on the framing technique. With our method, the memory can be saved and the efficiency can be improved to execute an MSVL program in MSV. In the future, the memory management of dynamic variables in functions will be considered. Besides, MSV toolkit will be further improved.

References

1. Aho, A.V., Hopcroft, J.E.: Design and Analysis of Computer Algorithms. Pearson Education India, Upper Saddle River (1974)
2. Arnold, K., Gosling, J., Holmes, D.: The Java Programming Language. Addison-Wesley, Reading (1996)
3. Barringer, H.: A Survey of Verification Techniques for Parallel Programs. LNCS, vol. 191. Springer, Berlin, New York (1985)
4. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
5. Duan, Z.: An extended interval temporal logic and a framing technique for temporal logic programming. Ph.D. thesis, University of Newcastle upon Tyne (1996)
6. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2005)
7. 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)
8. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Informatica **45**, 43–78 (2008)
9. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Sci. Comput. Program. **70**, 31–61 (2008)
10. Gay, D., Aiken, A.: Memory management with explicit regions. ACM SIGPLAN Not. **33**(5), 313–323 (1998)
11. Gosling, J.: The Java Language Specification. Addison-Wesley Professional, Boston (2000)
12. Kernighan, B.W., Ritchie, D.M., Ejeklint, P.: The C Programming Language. prentice-Hall, Englewood Cliffs (1988)
13. Prechelt, L., et al.: Comparing java vs. c/c++ efficiency differences to interpersonal differences. Commun. ACM **42**, 109–112 (1999)
14. Stroustrup, B., et al.: The C++ Programming Language. Pearson Education, India (1995)
15. Tofte, M., Talpin, J.-P.: Region-based memory management. Inf. Comput. **132**, 109–176 (1997)
16. Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D.: Dynamic storage allocation: a survey and critical review. In: Baler, H.G. (ed.) Memory Management. LNCS, vol. 986, pp. 1–116. Springer, Heidelberg (1995)
17. Zhisong, T., Chen, Z.: A temporal logic language oriented toward software engineering. J. Softw. **5**, 1–16 (1994)

Author Index