

Cong Tian  
Fumiko Nagoya  
Shaoying Liu  
Zhenhua Duan (Eds.)

LNCS 10795

# Structured Object-Oriented Formal Language and Method

7th International Workshop, SOFL+MSVL 2017  
Xi'an, China, November 16, 2017  
Revised Selected Papers

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, Lancaster, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Friedemann Mattern

*ETH Zurich, Zurich, Switzerland*

John C. Mitchell

*Stanford University, Stanford, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

C. Pandu Rangan

*Indian Institute of Technology Madras, Chennai, India*

Bernhard Steffen

*TU Dortmund University, Dortmund, Germany*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max Planck Institute for Informatics, Saarbrücken, Germany*

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

Cong Tian · Fumiko Nagoya  
Shaoying Liu · Zhenhua Duan (Eds.)

# Structured Object-Oriented Formal Language and Method

7th International Workshop, SOFL+MSVL 2017  
Xi'an, China, November 16, 2017  
Revised Selected Papers

*Editors*

Cong Tian  
Xidian University  
Xi'an  
China

Fumiko Nagoya  
Nihon University  
Tokyo  
Japan

Shaoying Liu  
Hosei University  
Koganei-shi, Tokyo  
Japan

Zhenhua Duan  
Xidian University  
Xi'an  
China

ISSN 0302-9743                      ISSN 1611-3349 (electronic)  
Lecture Notes in Computer Science  
ISBN 978-3-319-90103-9              ISBN 978-3-319-90104-6 (eBook)  
<https://doi.org/10.1007/978-3-319-90104-6>

Library of Congress Control Number: 2018940146

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© Springer International Publishing AG, part of Springer Nature 2018

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG  
part of Springer Nature  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

# Preface

The Structured Object-Oriented Formal Language (SOFL) provides an advanced technology to enhance the application of formal specification, specification-based testing and inspection, and specification animation for validation for industrial software development. Specifically, SOFL integrates data flow diagram, Petri nets, and VDM-SL to offer a visualized and formal notation for constructing specification; a three-step approach to requirements acquisition and system design; specification-based inspection and testing methods for detecting errors in both specifications and programs, and a set of tools to support modeling and verification. The Modeling, Simulation and Verification Language (MSVL) is a parallel programming language. Its supporting toolkit MSV has been developed to enable us to model, simulate, and verify a system in a formal manner.

Following the success of the previous SOFL+MSVL workshop, the 7th International Workshop on SOFL+MSVL was jointly organized in Xi'an on November 16, 2017 by Shaoying Liu's research group at Hosei University, Japan, and Zhenhua Duan's research group at Xidian University, China, with the aim of bringing together industrial, academic, and government experts and practitioners of SOFL or MSVL to communicate and to exchange ideas. Prof. Luke Ong of the University of Oxford was invited to give a keynote talk on "New Inductive Invariants for Concurrency." The workshop attracted 21 submissions on specification-based testing, specification inspection, model checking, formal verification, formal semantics, and formal analysis. Each submission was rigorously reviewed by two or more Program Committee members on the basis of technical quality, relevance, significance, and clarity, and 13 papers were accepted for publication in the workshop proceedings. The acceptance rate was 61.9%.

We would like to thank ICFEM 2017 for supporting the organization of the workshop, all of the Program Committee members for their great efforts and cooperation in reviewing and selecting the papers, and our postgraduate students for their help. We would also like to thank all of the participants for attending presentation sessions and actively joining discussions at the workshop. Finally, our gratitude goes to Springer for its continuous support in the publication of the workshop proceedings.

November 2017

Cong Tian  
Fumiko Nagoya  
Shaoying Liu  
Zhenhua Duan

# Organization

## Program Committee

Shaoying Liu (General Chair)	Hosei University, Japan
Zhenhua Duan (General Chair)	Xidian University, China
Cong Tian (Program Co-chair)	Xidian University, China
Fumiko Nagoya (Program Co-chair)	Nihon University, Japan
Guoqiang Li	Shanghai Jiao Tong University, China
Haitao Zhang	Lanzhou University, China
Hong Zhu	Oxford Brookes University, UK
Huaikou Miao	Shanghai University, China
Jing Sun	The University of Auckland, New Zealand
Karl Leung	Hong Kong Institute of Vocational Education, SAR China
Kazuhiro Ogata	JAIST, Japan
Richard Lai	La Trobe University, Australia
Shengchao Qin	Teesside University, UK
Shin Nakajima	National Institute of Informatics, Japan
Stefan Gruner	University of Pretoria, South African
Weikai Miao	East China Normal University, China
Wuwei Shen	Western Michigan University, USA
Xi Wang	Shanghai University, China
Xiaobing Wang	Xidian University, China
Xinfeng Shu	Xi'an University of Posts and Telecommunications, China
Yuting Chen	Shanghai Jiao Tong University, China

# Contents

## Animation and Prototyping

Graphically Perceiving Characteristics of the MCS Lock and Model Checking Them . . . . .	3
<i>Tam Thi Thanh Nguyen and Kazuhiro Ogata</i>	
An Investigation of Integrating a GUI-Aided Approach and a Specification-Based Testing . . . . .	24
<i>Fumiko Nagoya and Shaoying Liu</i>	

## Graph Theory

On the Cooperative Graph Searching Problem . . . . .	39
<i>Chin-Fu Lin, Ondřej Navrátil, and Sheng-Lung Peng</i>	

## Model Checking

Boosting UPPAAL for OSEK/VDX Applications with a Sequentialization Approach. . . . .	51
<i>Haitao Zhang, Zhuo Cheng, Jianxin Xue, and Yonggang Lu</i>	
The Complexity of Linear-Time Temporal Logic Model Repair . . . . .	69
<i>Xiuting Tao and Guoqiang Li</i>	
Extending UML for Model Checking . . . . .	88
<i>Xinfeng Shu, Mengnan Wang, and Xiaobing Wang</i>	

## Modeling and Specification

Foundation of a Framework to Support Compliance Checking in Construction Industry. . . . .	111
<i>Wuwei Shen, Guangyuan Li, Chung-Ling Lin, and Hongliang Liang</i>	
An Improved Reliability Testing Model Based on SOFL . . . . .	123
<i>Zhouxian Jiang, Honghui Li, and Xuetao Tian</i>	
A Framework Based on MSVL for Verifying Probabilistic Properties in Social Networks . . . . .	133
<i>Xiaobing Wang, Liyuan Ren, Liang Zhao, and Xinfeng Shu</i>	



Implementing MapReduce with MSVL . . . . . 148  
*Nan Zhang, Meng Wang, Zhenhua Duan,  
Cong Tian, and Jin Cui*

**Verification and Validation**

A Software Tool to Support the “Vibration” Method. . . . . 171  
*Pan Zhao and Shaoying Liu*

A Software Tool to Support Scenario-Based Formal Specification  
for Error Prevention . . . . . 187  
*Siyuan Li and Shaoying Liu*

A Proof Score Approach to Formal Verification of an Imperative  
Programming Language Compiler . . . . . 200  
*Dorian Daudier, Trinh Ngoc Quoc Bao, and Kazuhiro Ogata*

**Author Index** . . . . . 219

# **Animation and Prototyping**



# Graphically Perceiving Characteristics of the MCS Lock and Model Checking Them

Tam Thi Thanh Nguyen<sup>(✉)</sup>  and Kazuhiro Ogata<sup>(✉)</sup> 

School of Information Science,  
Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan  
{tamnguyen,ogata}@jaist.ac.jp

**Abstract.** The MCS list-based queuing lock (MCS) is a mutual exclusion protocol whose variants have been used in Java virtual machines. MCS is specified as a state machine in Maude, a rewriting logic-based computer language and system. We have developed a tool (called SMGA) that takes a finite computation generated from a state machine and displays its graphical animations. MCS is used to describe how such graphical animations help human beings perceive characteristics of the state machine of MCS. Such characteristics can be confirmed by Maude model checking facilities. The characteristics graphically perceived and confirmed by model checking could be used as lemmas to theorem prove that MCS enjoys some desired properties. SMGA can also display graphical animations of counterexamples presented by the Maude LTL model checker.

**Keywords:** Graphical animation · Maude · Model checking  
Mutual exclusion protocols · State machine

## 1 Introduction

State machines can be used as mathematical models of various systems and their properties can be used to formalize systems requirements. Thus, systems verification can be conducted as formal verification of state machine properties. Two major systems verification techniques are model checking and theorem proving. Model checking can be automatically conducted but cannot basically deal with infinite-state systems<sup>1</sup>. Theorem proving can directly deal with infinite-state systems but requires human interaction. One of the most intellectual tasks in theorem proving is conjecturing lemma.

We have developed a tool [3] that takes a finite computation of a state machine and displays its graphical animation. The tool (called the state machine

---

This work was partially supported by JSPS KAKENHI Grant Number 26240008.

<sup>1</sup> If you find a good abstraction that converts an infinite-state system to a finite-state one and preserves the negation of a property concerned, the infinite-state system can be formally verified by model checking the finite-state one [1, 2], although you need to prove the preservation of the negated property by the abstraction.

graphical animation tool, or the SMGA tool, or simply SMGA) mainly aims at helping human beings perceive characteristics appearing in state machine graphical animations and conjecture lemmas that could be used to theorem prove state machine properties. The MCS list-based queuing lock (the MCS protocol, the MCS lock, or simply MCS) [4] is a mutual exclusion protocol whose variants have been used in Java virtual machines. MCS is specified as a state machine in Maude [5], a rewriting logic-based computer language equipped with model checking facilities (the search command and the LTL model checker). SMGA takes finite computations from the Maude specification of MCS and displays their graphical animations. This paper describes how such graphical animations help human beings perceive characteristics of the state machine of MCS appearing in the animations. The Maude search command can be used to confirm the guessed characteristics by exhaustively traversing the Maude specification of MCS. If Maude refutes some, we can revise them based on the counterexamples generated by Maude. Characteristics perceived by human beings in graphical animations and confirmed by model checking would be likely to be able to be used as lemmas for theorem proving. The paper also describes model checking experiments that MCS enjoys the mutual exclusion property with the Maude search command and the lockout freedom property with the Maude LTL model checker. Two variants of MCS in which a complex atomic instruction `comp&swap` is not used are analyzed with the LTL model checker as well. One variant does not enjoy the lockout freedom property and then a counterexample is given by the model checker. SMGA can also generate a graphical animation of a counterexample.

The rest of the paper is organized as follows. Section 2 describes some preliminaries, such as Kripke structures and LTL. Section 3 describes MCS. Section 4 describes Maude and how specify MCS in Maude. Section 5 reports on the case study in which MCS and two variants have been analyzed with SMGA and Maude. Section 6 mentions some existing related work, and Sect. 7 finally concludes the paper.

## 2 Preliminaries

Let  $S$  be a set and  $\pi$  be an infinite sequence  $e_0; \dots; e_i; \dots$  of  $S$ , where each  $e_i \in S$ , and then  $\pi(i) = e_i$  (the  $i$ th element in  $\pi$ ) and  $\pi^i = e_i; \dots$  (the  $i$ th suffix obtained by deleting the first  $i$  elements from  $\pi$ ) for each natural number  $i$ . Let  $e_0; \dots; e_n$  be a non-empty finite sequence of  $S$ , and then  $(e_0; \dots; e_n)^\infty = e_0; \dots; e_n; e_0; \dots; e_n; \dots$  (the infinite sequence in which the finite sequence repeats infinitely often). Let  $U$  be a universal set of symbols.

A Kripke structure (KS)  $K$  is a 5 tuple  $\langle S, I, P, L, T \rangle$ , where  $S$  is a set of states,  $I \subseteq S$  is the set of initial states,  $P \subseteq U$  is a set of atomic state propositions,  $L$  is a labeling function whose type is  $S \rightarrow 2^P$ , and  $T \subseteq S \times S$  is a total binary relation. An element  $(s, s') \in T$  may be written as  $s \rightarrow s'$  and referred as a state transition.

A path of  $K$  is an infinite sequence  $s_0; \dots; s_i; s_{i+1}; \dots$  of  $S$  such that  $(s_i, s_{i+1}) \in T$  for each natural number  $i$ . A computation of  $K$  is a path  $\pi$  of

$K$  such that  $\pi(0) \in I$ . Let  $\mathcal{P}$  be the set of all paths of  $K$  and  $\mathcal{C}$  be the set of all computations of  $K$ . A finite prefix  $s_0; \dots; s_n$  of a computation (or path) of  $K$  is called a finite computation (or path) of  $K$ . The syntax of a formula  $\varphi$  in Linear Temporal Logic (LTL) for  $K$  is  $\varphi ::= \top \mid p \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \mathcal{U} \varphi$ , where  $p \in P$ . Let  $\mathcal{F}$  be the set of all formulas in LTL for  $K$ .

An arbitrary path  $\pi \in \mathcal{P}$  of  $K$  and an arbitrary LTL formula  $\varphi \in \mathcal{F}$  of  $K$ ,  $K, \pi \models \varphi$  is inductively defined as  $K, \pi \models \top$ ,  $K, \pi \models p$  if and only if  $p \in L(\pi(0))$ ,  $K, \pi \models \neg\varphi_1$  if and only if  $K, \pi \not\models \varphi_1$ ,  $K, \pi \models \varphi_1 \wedge \varphi_2$  if and only if  $K, \pi \models \varphi_1$  and  $K, \pi \models \varphi_2$ ,  $K, \pi \models \bigcirc\varphi_1$  if and only if  $K, \pi^1 \models \varphi_1$ , and  $K, \pi \models \varphi_1 \mathcal{U} \varphi_2$  if and only if there exists a natural number  $i$  such that  $K, \pi^i \models \varphi_2$  and for all natural numbers  $j < i$ ,  $K, \pi^j \models \varphi_1$ , where  $\varphi_1$  and  $\varphi_2$  are LTL formulas. Then,  $K \models \varphi$  if and only if  $K, \pi \models \varphi$  for each computation  $\pi \in \mathcal{C}$  of  $K$ .

The temporal connectives  $\bigcirc$  and  $\mathcal{U}$  are called the next operator and the until operator, respectively. The other logical and temporal connectives are defined as usual as follows:  $\perp \triangleq \neg\top$ ,  $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$ ,  $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$ ,  $\diamond\varphi \triangleq \top \mathcal{U} \varphi$ ,  $\square\varphi \triangleq \neg(\diamond\neg\varphi)$ , and  $\varphi_1 \rightsquigarrow \varphi_2 \triangleq \square(\varphi_1 \Rightarrow \diamond\varphi_2)$ . The temporal connectives  $\diamond$ ,  $\square$  and  $\rightsquigarrow$  are called the eventually operator, the always operator and the leadsto operator, respectively.

There are multiple possible ways to express states. In this paper, a state is expressed as an associative-commutative collection of name-value pairs, where a name may have parameters. Associative-commutative collections are called soups, and name-value pairs are called observable components. That is, a state is expressed as a soup of observable components. The juxtaposition operator is used as the constructor of soups. Let  $oc_1, oc_2, oc_3$  be observable components, and then  $oc_1 \ oc_2 \ oc_3$  is the soup of those three observable components. Since the order is irrelevant because of associativity and commutativity,  $oc_1 \ oc_2 \ oc_3$  is the same as some others, such as  $oc_3 \ oc_2 \ oc_1$ . For soups  $ocs_1, ocs_2$  of observable components,  $ocs_1 \subseteq ocs_2$  if and only if there exists a soup  $ocs_3$  of observable components such that  $ocs_1 \ ocs_3 = ocs_2$ , namely that there exists  $ocs_1$  in  $ocs_2$ , where each  $ocs_i$  for  $i = 1, 2, 3$  may be empty or a single observable component. Examples of observable components are (`glock: nop`) and (`pc[p1]: rs`), where `glock` and `pc[p1]` are names, `nop` and `rs` are values, and `p1` is a parameter of the name `pc[p1]`. An example of a soup of observable components is (`glock: nop`) (`pc[p1]: rs`) (`pc[p2]: rs`) (`pc[p3]: rs`). This represents (actually partially) a state in which there are three processes each of which is located at `rs` and there is one global variable `glock` that is shared by the three processes and whose value is `nop`. Since the soup is associative and commutative, even if some observable components are swapped, for example (`pc[p2]: rs`) (`pc[p1]: rs`) (`glock: nop`) (`pc[p3]: rs`), it represents the same state.

### 3 MCS List-Based Queuing Lock

The MCS list-based Queuing lock (MCS protocol) has been invented by Mellor-Crummey and Scott [4]. Variants of MCS protocol have been used in Java virtual

machines, and therefore the inventors were awarded the 2006 Edsger W. Dijkstra Prize in Distributed Computing<sup>2</sup>.

A pseudo-code of MCS protocol for each process  $p$  is as follows:

```

rs:  "Remainder Section"
11:   $next_p := \text{nop}$ ;
12:   $pred_p := \text{fetch\&store}(glock, p)$ ;
13:  if  $pred_p \neq \text{nop}$  {
14:     $lock_p := \text{true}$ ;
15:     $next_{pred_p} := p$ ;
16:    repeat while  $lock_p$ ; }
cs:  "Critical Section"
17:  if  $next_p = \text{nop}$  {
18:    if  $\text{comp\&swap}(glock, p, \text{nop})$ 
19:      goto rs;
110: repeat while  $next_p = \text{nop}$ ; }
111:  $locked_{next_p} := \text{false}$ ;
112: goto rs;

```

There is one global variable  $glock$  shared by all processes participating in MCS protocol. Its type is process IDs (or Pid). Initially,  $glock$  is  $\text{nop}$ , a dummy process ID. Each process  $p$  maintains three local variables  $next_p$ ,  $lock_p$  and  $pred_p$  whose types are Pid, Bool and Pid, respectively. Initially,  $next_p$ ,  $lock_p$  and  $pred_p$  are  $\text{nop}$ ,  $\text{false}$  and  $\text{nop}$ , respectively.  $next_p$  is used to construct a global queue of processes (or process IDs). Basically,  $next_p$  refers to the next element of the queue if  $p$  is in the queue. Since enqueueing an element into the queue and dequeuing the queue are not atomically done, however,  $next_p$  may be  $\text{nop}$  even though  $p$  is not the bottom element of the queue.  $pred_p$  refers to the previous element of the queue while  $p$  is being put into the queue.  $lock_p$  is the local lock on which process  $p$  is spinning while  $lock_p$  is true to wait for entering the critical section.  $glock$  basically refers to the bottom element if the queue is not empty. Since the two basic operations to the queue are not atomic, however,  $glock$  may not refer to the real bottom element while some process IDs are being put into the queue.

To safely conduct the two basic operations to the queue non-atomically, two atomic operations are used:  $\text{fetch\&store}$  and  $\text{comp\&swap}$ .  $\text{fetch\&store}(x, v)$  does the following atomically:  $tmp := x$ ,  $x := v$ , and  $tmp$  is returned, where  $tmp$  is a temporary variable.  $\text{comp\&swap}(x, v_1, v_2)$  does the following atomically: if  $x = v_1$ , then  $x := v_2$  and  $\text{true}$  is returned; otherwise,  $\text{false}$  is returned.

## 4 Maude

Maude is a rewriting logic-based computer language and system that is equipped with many functionalities, among which are model checking and meta-programming. Maude is one of the direct successors of OBJ3, the most

<sup>2</sup> <https://www.podc.org/dijkstra/2006-dijkstra-prize/>.

famous algebraic specification language and system mainly designed by Joseph A. Goguen. Therefore, Maude allows users to write specifications very flexibly. For example, associative and/or commutative binary operators can be freely used in specifications, making it possible to specify complex concurrent and distributed systems very succinctly.

As described, MCS protocol is formalized as a state machine whose states are expressed as soups of observable components. When there are three processes, a state is expressed as

```
(glock: G) (pc[p1]: L1) (pc[p2]: L2) (pc[p3]: L3) (next[p1]: P1)
(next[p2]: P2) (next[p3]: P3) (lock[p1]: B1) (lock[p2]: B2) (lock[p3]: B3)
(pred[p1]: Q1) (pred[p2]: Q2) (pred[p3]: Q3)
```

where  $G$ ,  $P_i$  and  $Q_i$  for  $i = 1, 2, 3$  are process IDs,  $L_i$  for  $i = 1, 2, 3$  are locations, such as *rs*, *l1* and *cs*, and  $B_i$  for  $i = 1, 2, 3$  are Booleans. Initially,  $G$ , each  $P_i$  and each  $Q_i$  are *nop*, each  $L_i$  is *rs*, each  $B_i$  is *false*. The initial state will be referred as *init*.

The state transitions are described in terms of rewrite rules as follows:

```
r1 [want] : (pc[P]: rs) => (pc[P]: l1) .
r1 [stnxt] : (pc[P]: l1) (next[P]: Q) => (pc[P]: l2) (next[P]: nop) .
r1 [stprd] : (glock: Q) (pc[P]: l2) (pred[P]: Q1)
=> (glock: P) (pc[P]: l3) (pred[P]: Q) .
r1 [chprd] : (pc[P]: l3) (pred[P]: Q)
=> (pc[P]: (if Q == nop then cs else l4 fi)) (pred[P]: Q) .
r1 [stlck] : (pc[P]: l4) (lock[P]: B) => (pc[P]: l5) (lock[P]: true) .
r1 [stnpr] : (pc[P]: l5) (pred[P]: Q) (next[Q]: Q1)
=> (pc[P]: l6) (pred[P]: Q) (next[Q]: P) .
r1 [chlck] : (pc[P]: l6) (lock[P]: false) => (pc[P]: cs) (lock[P]: false) .
r1 [exit] : (pc[P]: cs) => (pc[P]: l7) .
r1 [rpnxt] : (pc[P]: l7) (next[P]: Q) => (pc[P]: (if Q == nop then l8
else l11 fi)) (next[P]: Q) .
r1 [chglk] : (glock: Q) (pc[P]: l8) => (glock: (if Q == P then nop
else Q fi)) (pc[P]: (if Q == P then l9 else l10 fi)) .
r1 [go2rs] : (pc[P]: l9) => (pc[P]: rs) .
cr1 [rpnxt2] : (pc[P]: l10) (next[P]: Q) => (pc[P]: l11)
(next[P]: Q) if Q /= nop .
r1 [stlnx] : (pc[P]: l11) (next[P]: Q) (lock[Q]: B)
=> (pc[P]: l12) (next[P]: Q) (lock[Q]: false) .
r1 [gotrs] : (pc[P]: l12) => (pc[P]: rs) .
```

where *want*, *stnxt*, etc. are the labels of the rewrite rules.

## 5 Analysis of MCS Protocol

### 5.1 Invariant Model Checking with Search

For a state machine specification in Maude, a state  $S$ , a pattern  $P$  and a condition  $C$ , the Maude search command exhaustively traverses the reachable states from  $S$  to find states that match  $P$  and satisfy  $C$ :

```
search [N] in Mod : S =>* P such that C .
```

where  $N$  is a natural number. The search command tries to find at most  $N$  solutions. Note that a solution is basically a state  $A$  that matches  $P$  and satisfies  $C$ , but since there may be more than one substitution  $\sigma$  such that  $\sigma(P) = A$ , there may be more solutions than the number of such states and such substitutions are called solutions of the search.

The mutual exclusion property that should be enjoyed by mutual exclusion protocols, such as MCS protocol, says that there exists at most one process in the critical section at any given moment. Therefore, the search command can be used to check if MCS protocol enjoys the property as follows:

```
search [1] in MCS-INIT :
init =>* (pc[I]: cs) (pc[J]: cs) S .
```

where `MCS-INIT` is the module in which MCS protocol is specified in Maude,  $I$  and  $J$  are Maude variables of process IDs, and  $S$  is a Maude variable of states (or soups of observable components). If Maude finds a solution, MCS protocol does not enjoy the property. Maude did not find any solutions, implying that MCS protocol enjoys the property when there are three processes.

## 5.2 Graphical Animations of MCS Protocol

The graphical animation tool [3] has been implemented with DRAW-SVG [6] that is designed and developed by Joseph LIARD. It is a free online drawing application for designers and developers, making it possible to create fully standard compliant SVG. We have used DRAW-SVG as an integrated drawing tool within our tool to support users draw SVG pictures for any state machines. Our tool is available on the website <https://tamntt.bitbucket.io/Research/GraphicalAnimation/>. It allows users to design their own pictures of animations. Figure 1 shows the picture we have drawn for MCS protocol when there are three processes.

The graphical animation tool does not deal with state machines themselves internally. Instead, what is fed into the tool is basically a finite computation of a state machine. An example input file of  $M_{MCS}$  is as follows:

```
###keys
glock pc[p1] pc[p2] pc[p3] next[p1] next[p2] next[p3] lock[p1] lock[p2]
lock[p3] pred[p1] pred[p2] pred[p3]

###textDisplay

###states
(glock: nop (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (next[p1]: nop)
(next[p2]: nop) (next[p3]: nop) (lock[p1]: false) (lock[p2]: false)
(lock[p3]: false) (pred[p1]: nop) (pred[p2]: nop) pred[p3]: nop) ||
(glock: nop (pc[p1]: rs) (pc[p2]: l1) (pc[p3]: rs) (next[p1]: nop)
(next[p2]: nop) (next[p3]: nop) (lock[p1]: false) (lock[p2]: false)
```



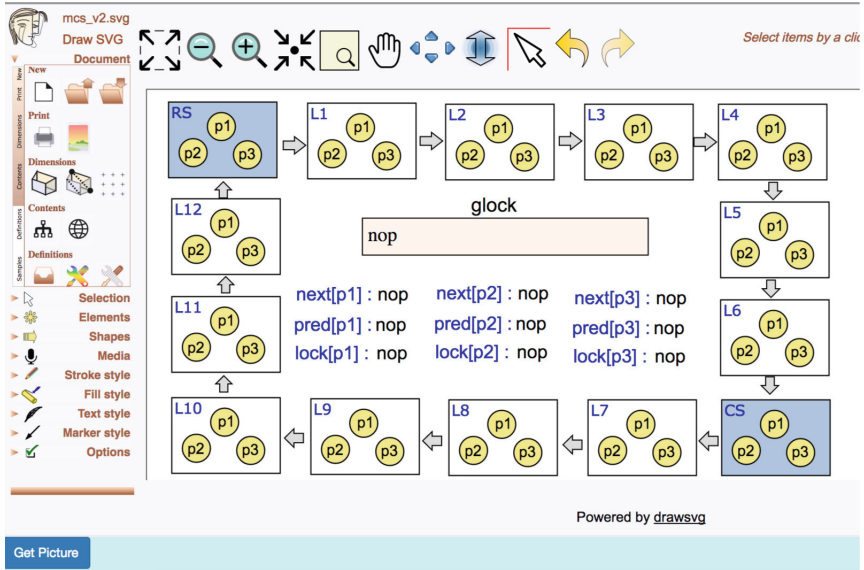


Fig. 1. Picture of MCS protocol

```
(lock[p3]: false) (pred[p1]: nop) (pred[p2]: nop) pred[p3]: nop ||
glock: nop (pc[p1]: 11) (pc[p2]: 11) (pc[p3]: rs) (next[p1]: nop)
(next[p2]: nop) (next[p3]: nop) (lock[p1]: false) (lock[p2]: false)
(lock[p3]: false) (pred[p1]: nop) (pred[p2]: nop) pred[p3]: nop
```

There are three segments in an input file as follows:

- *###keys*: This is a list of keys which are names of observable components in a state. The order in which the keys appear must be the same as the order in which the corresponding observable components appear in each state.
- *###textDisplay*: This part specifies how the value of an observable component is displayed. If nothing is specified, it is displayed horizontally and its top appears left most when displaying a queue or string list. There may be the case, however, where its top should appear right most. Some values, such as stacks, may have to be displayed vertically instead. The format used in this part is as follows:

```
key:::option:::regex(0)++++...++++regex(i)
```

The format consists of three parts: key, option and regex. A key appearing in the key segment is written in the key part. REV, VER or VER-REV is written in the option part. REV specifies a collection, such as queues and lists, is displayed such that its top appears right most, VER specifies a collection, such as stacks, is displayed vertically such that its top appears top most,

and VER-REV specifies a collection is displayed vertically such that its top appears bottom most. A list of regular expressions is written in the regexs part. For example, we have an observable component in a state as (*chan1* : < *false, pac(1)* > < *true, pac(2)* > *empty*), and we want the tool will display value of *chan1* as *empty* < *true, pac(2)* > < *false, pac(1)* >, the textDisplay segment is as follows:

```
chan1:::REV:::<_,_>++++empty
```

Two regular expressions <\_,\_> and **empty** are written in the regexs part. They match texts, such as <false,p(1)>, <true,p(2)>, and **empty**. For the case *MCS*, nothing is specified in the **###textDisplay** part since values of observable components are displayed horizontally and theirs top appear left most.

- **###states**: This is a finite computation of a state machine, namely a finite sequence of states. The sign || is a separator used to distinguish adjacent states.

After drawing the picture of a state machine, the user needs to edit properties for texts on the picture so that the observable components of the state machine can appear on the picture when the state machine is animated. As clicking a text on the picture and choosing the icon of properties, a pop-up will be displayed for editing properties. In this pop-up, the *name* as an ID for the text of an observable component (*name* : *value*) is set for the text so that the *value* can be displayed at the place where the text is located. The ID will be used for mapping it to the values whose name is *name* appearing in an input data when we run the graphical animation tool. For example, Fig. 2 shows *glock* is set as the ID of the observable component (*glock* : *nop*) so that the *nop* is displayed at the designated place on a state machine picture.

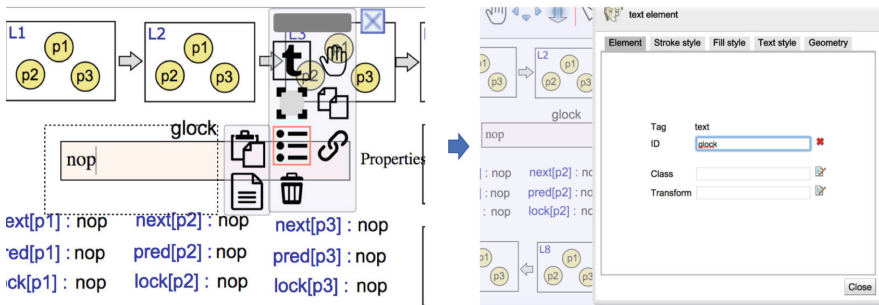
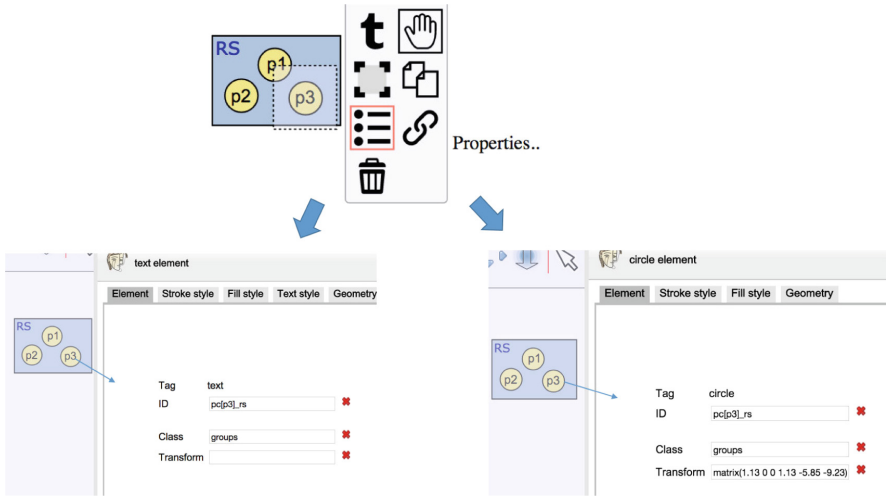


Fig. 2. Setting the property ID for displaying values of *glock* of  $M_{MCS}$

On the other hand, we want to display (*name* : *value*) pairs at different locations such as process *p1* at *cs*, process *p2* at *rs*, . . . . Thus, we can draw

SVG elements as rectangles to display for locations such as  $rs$ ,  $cs$ ,  $\dots$ , and draw circles with texts for displaying every processes for every location. Then, we will set properties for the circle and the text of every process at every location. The property *class* of them will be also set is *groups*. And the property *ID* of the circle, and the text of every process will be set as structure *KEY\_VALUE*, where *KEY* is the name, and *VALUE* is the value of a name-value pair. By this way, we can see that locations of processes are changed and displayed graphically when the tool animate states. For example, Fig. 3 shows how to set properties for the process  $p3$  at location  $rs$ . To display the process  $p3$  at the location  $rs$ , we will set the property *ID* is  $pc[p3]_{rs}$  for both the circle, and text element which visualize process  $p3$ . And we will also set the property *class* is *groups* for them.



**Fig. 3.** Setting properties such as *Class*, *ID* for displaying the process  $p3$  at the location  $rs$  of  $M_{MCS}$

After getting a drawn picture of a state machine and importing a prepared input file, the tool can run to play a graphical animation of the state machine. The tool allows human users to adjust the duration of the speed of animation. The unit of duration is millisecond. The smaller the duration is, the faster the animation is played. Animations can be played step by step in addition to that they can be played automatically from the beginning to the end. When an animation is played step by step, we can observe each state transition graphically.

The graphical animation tool basically takes a sequence of states and plays it graphically. The main purpose of the tool is to help human users recognize some useful patterns in animated computations, and therefore it is necessary to generate a long sequence of states. The Maude search command can generate sequences of states, but cannot generate very long sequences, such as a sequence that consists of 100 or more states due to the state explosion problem. Thus,

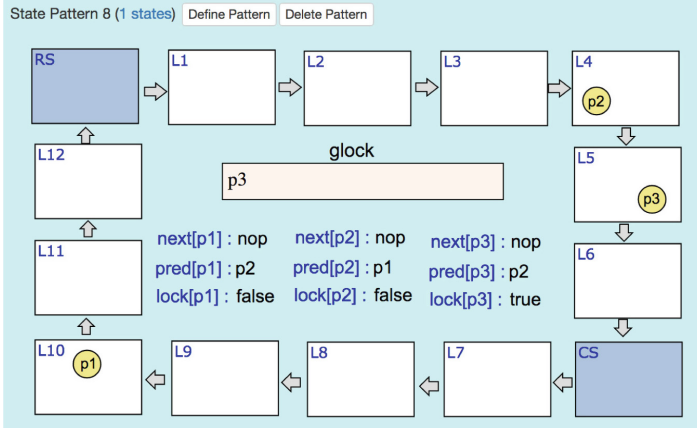


Fig. 4. A state such that  $p_1$  is at l10

we had written a meta-program in Maude to generate a long sequence of states. We used the meta-program to generate a finite computation that consists of 200 states for MCS protocol.

The tool can select and display the states that satisfy a condition from the input finite computation. The format of a defined condition is as follows:

```
(state['key1'] op1 state['key2']) op2 (state['key3'] op4 'value') ...
```

where  $key_1$ ,  $key_2$ , and  $key_3$  are names of observable components in states and keys appearing in the key segment of an input file,  $op_1$ ,  $op_2$ , and  $op_3$  are JavaScript comparison and logical operators, and  $value$  is a value. We asked the tool to select and display the states such that the location of  $p_1$  is 110 by using a condition that is defined as  $(state[pc[p_1]] == 'l10')$ . The tool found 16 such states in the input finite computation. Figure 4 shows one of the 16 states. The tool lets us know the state appear in the input finite computation at position 153. In the state, since  $p_1$  is at l10,  $p_1$  is dequeuing the global queue, while since  $p_2$  and  $p_3$  are 14 and 15,  $p_2$  and  $p_3$  are enqueueing  $p_2$  and  $p_3$  into the global queue, respectively, but none of them has completed. Given a state number  $n$ , the tool displays the state at position  $n$ . We asked the tool to display the state at position 153 and play the animation from the state step by step. Figure 5 shows the five states at positions 154, 155, 156, 157 and 158 from the top. In state 153,  $p_2$  executes the assignment at l4, setting  $lock_{p_2}$  true, and moves to l5 but has not yet completed enqueueing  $p_2$  into the global queue. In state 154,  $p_2$  executes the assignment at l5, setting  $next_{p_1}$  to  $p_2$ , and moves to l6, when  $p_2$  has eventually completed enqueueing  $p_2$  into the global queue. In state 155,  $glock$  is  $p_3$ , meaning that  $p_3$  is the bottom element of the global queue but  $p_3$  has not completed enqueueing  $p_3$  into the global queue. In state 155,  $p_1$  leaves the loop at l10 and moves to l11 but has not yet completed dequeuing the global queue. In state 156,  $p_3$  executes the assignment at l5, setting  $next_{p_2}$  to  $p_3$ , and moves

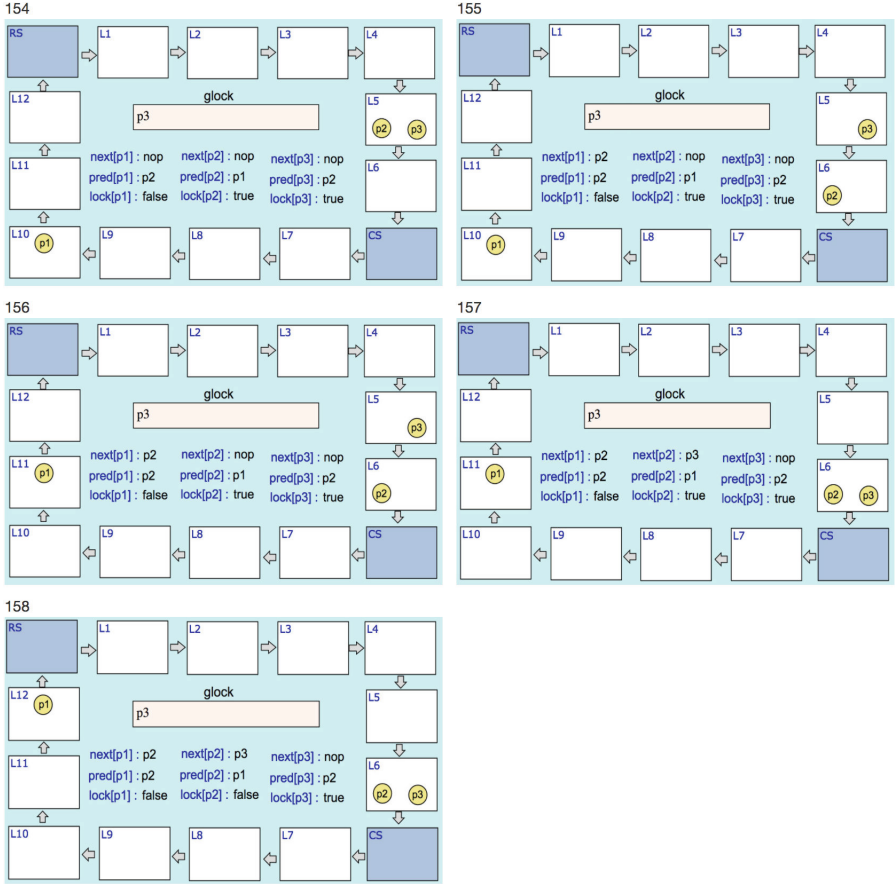


Fig. 5. States 154, 155, 156, 157 and 158

to l6, when p3 has eventually completed enqueueing p3 into the global queue. In state 157, p1 executes the assignment at l11, setting  $lock_{p2}$  false, letting know p2 is ready to enter the critical section, and moves to l12. In state 158, the global queue consists of p2 and p3 in this order because  $lock_{p2}$  is false,  $next_{p2}$  is p3,  $lock_{p2}$  is true,  $next_{p2}$  is nop, and glock is p3.

### 5.3 Perceiving Characteristics with Graphical Animations

By observing graphical animations of *MCS*, we have also found some characteristics or patterns appearing in them. Although we do not prove those characteristics in this paper, we will model check them in the next sub-section. Proving the characteristics is one piece of our future work. In this sub-section, we present some characteristics guessed by observing graphical animations of a finite computation *FC1000* that consists of 1000 states as follows:

*Characteristic 1:*

If there is a process in the critical section *cs*, the local lock owned by each process that wants to enter the *cs* is true.

*Characteristic 2:*

If a process *p* is at the location *l3* and  $pred[p] = nop$ , there is no process in the critical section *cs*.

*Characteristic 3:*

If a process *p* is at the location *l6* and  $lock[p] = false$ , there is no process in the critical section.

*Characteristic 4:*

If a process *p1* is at the location *l12*, another process *p2* is at *l6*, and  $pred[p2] = p1$  then the  $lock[p2]$  is *false*.

*Characteristic 5:*

If a process *p1* is at the location *l9* and another process *p2* is at *l4* then the *glock* is *p2*.

*Characteristic 6:*

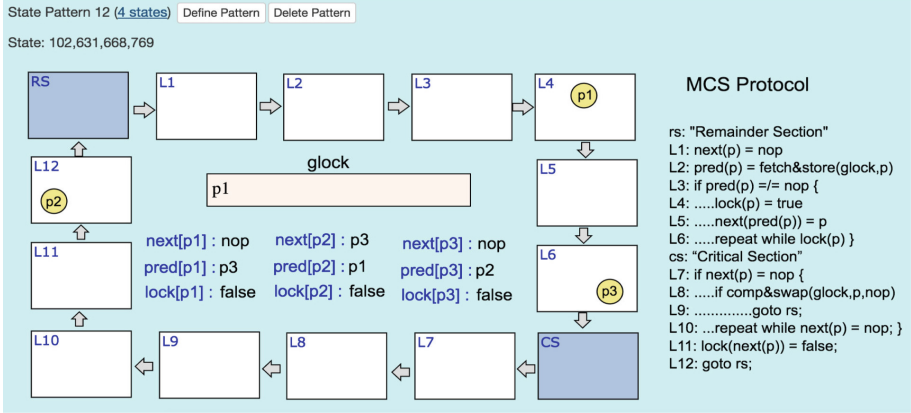
No state such that a process is at *cs*, *l7*, *l8*, *l10*, or *l11* and another state is at *cs*, *l7*, *l8*, *l10*, or *l11*.

*Characteristic 7:*

If there is a process is at *l3*, *l4*, *l5*, *l6*, *cs*, *l7*, *l8*, *l10*, or *l11*, *glock*  $\neq nop$ .

Besides taking a close look at several graphical animations of MCS to recognize some characteristics appearing in the graphical animations, we can also ask the tool to show us all states that satisfy some conditions. The tool supports us to select the states among the ones in a given input file such that a condition is fulfilled and to display their graphical representations. If some states are similar each other, they will be clustered into one state representation as a state pattern. For example, if we have a sequence of state such as A, B, C, A, D, E, B, . . . , the tool will group same states, and display different states such as A, B, C, D, E. Thus, we can define some conditions to filter satisfied states to check or confirm some predicted characteristics, and reduce the amount of time for animations observing. If the tool refutes guessed characteristics, we should correct them. An example (called *Cond1*) of the conditions is as follows:

```
((state['pc[p1]'] == 'l12' ) || (state['pc[p2]'] == 'l12' ) ||
(state['pc[p3]'] == 'l12' )) && ((state['pc[p1]'] == 'l6' )
|| (state['pc[p2]'] == 'l6' ) || (state['pc[p3]'] == 'l6' ))
```



**Fig. 6.** A state pattern clustered from four same states, which satisfies the condition *Cond1*.

This condition can select the states such that there is a process  $p1$  in the  $l12$ , and there is another process  $p2$  in the  $l6$ . By using the *Cond1* for selecting satisfied states from an input file in which the states segment is *FC1000*, we found 130 states clustered into 66 state patterns. Figure 6 shows a state pattern that satisfies the condition *Cond1*. This pattern is a representation of four same states such as the state 102, 631, 668, and 769. We used *Cond1* to check a guessed characteristic (called *Pre4*) as follows:

If there is a process  $p1$  in the  $l12$ , and there is another process  $p2$  in the  $l6$  then  $lock[p2] = false$ .

However, we found some states do not satisfy characteristic *Pre4*. One of them is shown in Fig. 7. In this state, process  $p2$  is at  $l12$ , two processes  $p1$  and  $p3$  are at  $l6$ ,  $lock[p1] = true$ ,  $lock[p3] = false$ . Thus, we reviewed the graphically displayed states that enjoyed *Cond1*. And we perceived that if a process  $p1$  is at the location  $l12$ , another process  $p2$  is at  $l6$ , and  $pred[p2] = p1$  then the  $lock[p2]$  is *false*. This is the *Characteristic4* which we have mentioned above. To check our guess, we made another condition (called *Cond2*) as follows:

```
((state['pc[p1]'] == 'l12') && (((state['pc[p2]'] == 'l6') &&
(state['pred[p2]'] == 'p1')) || ((state['pc[p3]'] == 'l6') &&
(state['pred[p3]'] == 'p1')))) || ((state['pc[p2]'] == 'l12')
&& (((state['pc[p1]'] == 'l6') && (state['pred[p1]'] == 'p2'))
|| ((state['pc[p3]'] == 'l6') && (state['pred[p3]'] == 'p2'))))
|| ((state['pc[p3]'] == 'l12') && (((state['pc[p2]'] == 'l6')
&& (state['pred[p2]'] == 'p3')) || ((state['pc[p1]'] == 'l6')
&& (state['pred[p1]'] == 'p3'))))
```

This condition can select the states such that there is a process  $p1$  in the  $l12$ , and there is another process  $p2$  in the  $l6$  such that  $pred[p2] = p1$ . We found 111

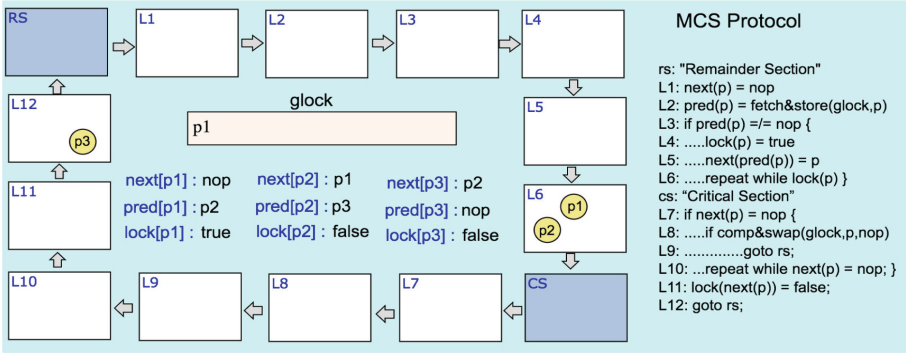


Fig. 7. A state does not satisfy the characteristic *Pre4*.

states clustered into 51 state patterns, and perceived that all of them satisfied *Characteristic4*.

Thus, the tool supports us usefully to perceive some characteristics appearing graphical animations. When paired with an intuitively designed picture and sequence of states, the tool can create well-crafted animated visualizations which are effective at attracting viewers and supporting them to more conveniently understand complex characteristics. After using graphical animations to get better understandings of  $M_{MCS}$  and recognizing some characteristics, we will model check the characteristics by Maude to confirm them. Some of them may be refuted by model checking, which allows human beings to revise such characteristics based on the counterexamples given by a model checker and may help them get better understandings of  $M_{MCS}$ .

#### 5.4 Confirming Characteristics with Maude

In this sub-section, we describe how to confirm the characteristics with the search command in Maude.

We asked the Maude search command to find a state such that a given characteristic (or property) is broken. If Maude finds a solution, the characteristic is not enjoyed by *MCS* protocol. Otherwise, *MCS* enjoys the characteristic (or the property) when there are three processes because we use them in the model checking experiments. The seven characteristics guessed with the help of the state machine graphical animation tool can be confirmed with the Maude search command:

– Characteristic 1:

```

search [1] in MCS-INIT : init =>* (pc[I]: L1) (pc[J]: L2) (lock[J]: B)
S such that not ((L1 == cs and (L2 == l6)) implies B) .
    
```



– Characteristic 2:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pred[I]: K) (pc[J]: L2)
S such that not ((L1 == 13 and K == nop) implies not (L2 == cs)) .
```

– Characteristic 3:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (lock[I]: B) (pc[J]: L2)
S such that not ((L1 == 16 and not B) implies not (L2 == cs)) .
```

– Characteristic 4:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pc[J]: L2) (lock[J]: B)
(pred[J]: K) S such that not ((L1 == 112 and L2 == 16 and K == I)
implies not B) .
```

– Characteristic 5:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pc[J]: L2) (glock: K)
S such that not (((L1 == 19) and (L2 == 14)) implies (K == J)) .
```

– Characteristic 6:

```
search [1] in MCS-INIT : init =>* (pc[I]: L1) (pc[J]: L2) S such that
(L1 == cs or L1 == 17 or L1 == 18 or L1 == 19 or L1 == 110 or
L1 == 111) and (L2 == cs or L2 == 17 or L2 == 18 or L1 == 19 or
L2 == 110 or L2 == 111) .
```

– Characteristic 7:

```
search [1] in MCS-INIT : init =>* (glock: K) (pc[I]: L) S such that
not ((L == 13 or L == 14 or L == 15 or L == 16 or L == cs or L == 17
or L == 18 or L == 110 or L == 111) implies (not K == nop)) .
```

where MCS-INIT is the module in which MCS protocol is specified in Maude, I, J, and K are Maude variables of process IDs, L, L1, and L2 are Maude variables of locations, B is a Maude variable of Boolean, and S is a Maude variable of states (or soups of observable components). Each of the seven search commands found no solution, meaning that MCS enjoys the seven characteristics (or the seven properties) when there are three processes. The model checking experiments, however, do not guarantee that MCS enjoys the seven characteristics for an arbitrary number of processes. We will theorem prove the seven characteristics for an arbitrary number of processes in future by writing what are called proof scores in CafeOBJ, a sibling language of Maude. We predict that some of the seven characteristics could be used as lemmas when we theorem prove that MCS enjoys the mutual exclusion property for an arbitrary number of processes.

### 5.5 LTL Model Checking

In this sub-section and the following sub-sections, we suppose that there are two processes `p1` and `p2` and let `init` denote the initial state in which the two processes participate in MCS protocol.

To use Maude LTL model checker, users are supposed to specify atomic propositions. Let us suppose we model check MCS protocol enjoys the lockout



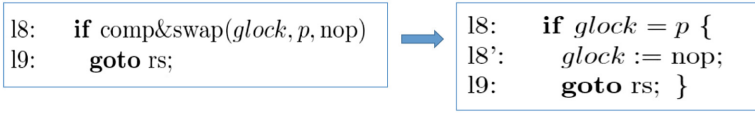
**Fig. 8.** A counterexample for the lockout freedom property for MCS protocol in which `comp&swap` is not naively used.



If we use `chlck'` and `rpnxt2'` instead of `chlck` and `rpnxt2`, reducing `modelCheck(init, lofuf(p1))` generates a counterexample. This is because the assumption used does not prohibit a process only repeats a loop forever. Each of the loops used in MCS protocol does not change anything if its condition is true. Therefore, the two loops can be formalized as the rewrite rules `chlck` and `rpnxt2`, which could make the assumption simpler.

## 5.6 A Naive Way to Disuse `comp&swap`

MCS protocol uses two atomic operators `fetch&store` and `com&swap`. We model check the two properties for a variant of MCS protocol in which `comp&swap` is not used. Figure 9 shows how to change the protocol.



**Fig. 9.** A modification such that `comp&swap` is disused.

The rewrite rule `chglk` is then replaced with the following two rewrite rules:

```
r1 [chglk'] : (glock: Q) (pc[P]: l8) => (glock: Q) (pc[P]: (if Q == P
                                     then l8' else l10 fi)) .
r1 [stglk] : (glock: Q) (pc[P]: l8') => (glock: nop) (pc[P]: l9) .
```

Model checking the two properties for the variant, the search command does not find any counterexamples for the mutual exclusion property but the LTL model checker finds a counterexample for the lockout freedom property even if a fair scheduler is adopted. Note that we can use exactly the same assumption used to model check that MCS protocol enjoys the lockout freedom property.

A counterexample generated by Maude LTL model checker consists of a finite computation from an initial state to a state leading to an infinite loop such that a finite state sequence is repeated forever or leading to a deadlock state. The counterexample generated by Maude LTL model checker for the lockout freedom under the use of a fair scheduler consists of a finite computation that consists of 17 states leading to an infinite loop such that a finite state sequence that consists of 9 states is repeated forever. We had extended the state machine graphical animation tool so that a counterexample can be graphically animated [7]. Feeding the counterexample generated by Maude LTL model checker, the extended tool graphically animates it, repeating the loop part, which lets us realize only `p2` enters and leaves the critical section repeatedly while `p1` is waiting at `l6` until `lockp1` becomes false. Figure 8 shows the 26 pictures of the states composing the counterexample. The first 17 states is the finite computation, while the last 9

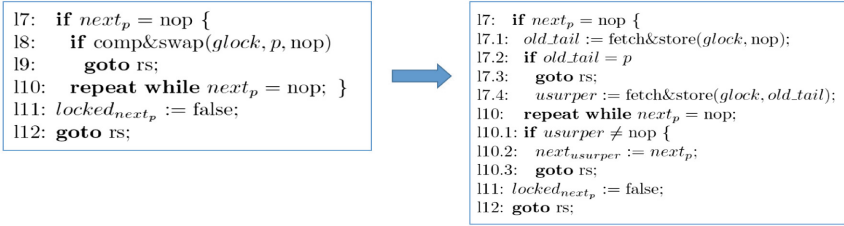


Fig. 10. A correction of the wrong part.

states is the finite state sequence that repeats forever, making the loop. Note that state 0 is the top state of the finite computation.

In state 8,  $p_1$  is at l2 and is enqueueing it into the global queue, and  $p_2$  is at l8 and is dequeuing the global queue.  $p_2$  checks the condition of the if statement at l8. Since  $glock$  is not  $p_2$ ,  $p_2$  moves to l8'. In state 9,  $p_1$  executes  $pred_{p_1} := fetch\&store(glock, p_1)$ ; at l2, making  $glock$   $p_1$  and  $pred_{p_1}$   $p_2$ . In state 9, since  $pred_{p_1}$  is  $p_2$ , the predecessor of  $p_1$  is  $p_2$  in the global queue, meaning that  $p_1$  has not been extracted from the global queue. In what follows, since  $pred_{p_1}$  is not  $nop$ ,  $p_1$  sets  $next_{p_2}$  to  $p_1$  and  $lock_{p_1}$  true, and waits at l6 until  $lock_{p_1}$  becomes false. In state 13,  $p_2$  executes  $glock := nop$ ; at l8'. Therefore, in state 14,  $glock$  is  $nop$ , meaning that the global queue is empty, although  $p_1$  is waiting at l6 until  $lock_{p_1}$  becomes false. This is way  $p_1$  is waiting at l6 forever and only  $p_2$  enters and leaves the critical section repeatedly.

## 5.7 The Mellor-Crummey and Scott's Way to Disuse $comp\&swap$

Mellor-Crummey and Scott have also proposed how to implement MCS protocol such that  $comp\&swap$  is not used. Figure 10 shows their way to disuse  $comp\&swap$ .

Accordingly, the Maude specification of MCS has been revised, and model checking for the lockout freedom property has been conducted. No counterexample was found.

## 6 Related Work

MCS protocol has been formally verified. Wang [8] has automatically conducted formal proof that MCS protocol enjoys the mutual exclusion property for an arbitrary number of processes but has not for the lockout freedom property. Wang and Schmidt [9] have proposed a way to formally conduct symmetric symbolic safety-analysis of concurrent software with pointer data structures. MCS protocol has been used as an example to demonstrate the proposed technique. They only consider the mutual exclusion property but not the lockout freedom property. The second author of the present paper and Futatsugi [10] have semi-formally proved that MCS protocol enjoys both the mutual exclusion property and the lockout freedom property. Neither of them has taken into account the

two variants of MCS protocol in which `comp&swap` is naively disused and the Mellor-Crummey and Scott's way to disuse of `comp&swap` is used. Schellhorn et al. [11] have proposed a way to prove concurrent programs enjoy the lockout (or starvation) freedom property. They proved MCS protocol as an example enjoys the property. However, none of them has graphically animated MCS protocol.

Most formal specification languages, such as Z, B method, and Event-B, are not executable, although some, such as VDM and VDM++, are semi-executable. Therefore, some researches have been carried out, making formal specifications written in such languages run, for example, by translating sub-sets of such languages into programming languages. Running formal specifications is called specification animation. Specification animation makes it possible to help human users get better understandings of formal specifications. Therefore, specification animation have been used to improve some other activities, such as refinement [12, 13], inspection and formal specification construction [14, 15], and software monitoring [16]. Although specification animation does not necessarily mean visual and graphical animations, some tools make it possible to play graphical animations [15]. The formal specification language we have used is Maude. Since Maude is executable, we do not need to develop any translators. Our approach to use of Maude and the state machine graphical animation tool has been directing a similar goal to those of these existing studies.

The second author of the present paper and Futatsugi have semi-formally proved that MCS protocol enjoys both the mutual exclusion property and the lockout freedom property, but have not formally proved. The semi-formal proofs may have overlooked several subtle lemmas. The main purpose of the state machine graphical animation tool helps human users recognize some useful patterns from graphically animated computations. from which human users could conjecture useful lemmas [3]. One piece of our future work is to recognize useful patterns from several graphically animated computations, conjecture useful lemmas from the animated computations and formally verify MCS protocol enjoys the mutual exclusion property and the lockout freedom property.

## 7 Conclusion

MCS was used to demonstrate how graphical animations of the state machine of MCS help human beings perceive characteristics of the state machine appearing in the animations. Graphical animations of a state machine are generated by SMGA from finite computations of the state machine. Such guessed characteristics can be confirmed by the Maude search command. If a counterexample is found for a guessed characteristic, we could revise the characteristic based on the counterexample. The characteristics graphically perceived and confirmed could be used as lemmas to theorem prove that MCS enjoys desired properties, such as the mutual exclusion property and the lockout freedom property. We also described the model checking experiments with the Maude LTL model checker that MCS and two variants enjoy the mutual exclusion property.

## References

1. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* **16**, 1512–1542 (1994)
2. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. *Theor. Comput. Sci.* **403**, 239–264 (2008)
3. Nguyen, T.T.T., Ogata, K.: Graphical animations of state machines. Submitted for Publication (2017). <https://tamntt.bitbucket.io/Research/Paper/smga.pdf>
4. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* **9**, 21–65 (1991)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71999-1>
6. Liard, J.: Draw SVG Website (2015). <http://www.drawsvg.org/>
7. Nguyen, T.T.T., Ogata, K.: A way to comprehend counterexamples generated by Maude LTL model checker. Submitted for Publication (2017)
8. Wang, F.: Automatic verification of pointer data-structure systems for all numbers of processes. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 328–347. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-48119-2\\_20](https://doi.org/10.1007/3-540-48119-2_20)
9. Wang, F., Schmidt, K.: Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 50–64. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-36135-9\\_4](https://doi.org/10.1007/3-540-36135-9_4)
10. Ogata, K., Futatsugi, K.: Formal verification of the MCS list-based queuing lock. In: Thiagarajan, P.S., Yap, R. (eds.) ASIAN 1999. LNCS, vol. 1742, pp. 281–293. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-46674-6\\_24](https://doi.org/10.1007/3-540-46674-6_24)
11. Schellhorn, G., Travkin, O., Wehrheim, H.: Towards a thread-local proof technique for starvation freedom. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 193–209. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-33693-0\\_13](https://doi.org/10.1007/978-3-319-33693-0_13)
12. Hallerstede, S., Leuschel, M., Plagge, D.: Refinement-animation for event-B – towards a method of validation. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 287–301. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-11811-1\\_22](https://doi.org/10.1007/978-3-642-11811-1_22)
13. Hallerstede, S., Leuschel, M., Plagge, D.: Validation of formal models by refinement animation. *Sci. Comput. Program.* **78**, 272–292 (2013)
14. Liu, S.: Validating formal specifications using testing-based specification animation. In: FormaliSE@ICSE 2016, pp. 29–35 (2016)
15. Li, M., Liu, S.: Integrating animation-based inspection into formal design specification construction for reliable software systems. *IEEE Trans. Reliabil.* **65**, 88–106 (2016)
16. Liang, H., Dong, J.S., Sun, J., Wong, W.E.: Software monitoring through formal specification animation. *ISSE* **5**, 231–241 (2009)



# An Investigation of Integrating a GUI-Aided Approach and a Specification-Based Testing

Fumiko Nagoya<sup>1</sup>(✉) and Shaoying Liu<sup>2</sup>

<sup>1</sup> College of Commerce, Nihon University, Tokyo, Japan  
nagoya.fumiko@nihon-u.ac.jp

<sup>2</sup> Faculty of Computer and Information Sciences, Hosei University, Tokyo, Japan  
sliu@hosei.ac.jp

**Abstract.** Formal methods mainly aim to improve software reliability by systematic refinements and/or verifications between specifications and implementations in mathematical ways. However, they are not enough for validating whether the system functions meet the user's requirements. We propose a combination of software prototyping and formal methods to address this problem. Our prototype provides desirable behaviors of system functions, and supports the generation of test cases by finding out input and output data related to the functions. This paper describes a case study for integrating a GUI-aided approach to constructing formal specifications and a testing-based verification of programs. Our research shows a demand for additional work on verifying external database accesses in automatic test case generation from formal specifications.

## 1 Introduction

Software testing is a practical way to evaluate software products by observing its execution, even though it has a fundamental limitation [1]; testing can only show the presence of errors, never their absence. Regarding quality assurance of software, expensive costs and labor intensive works are inevitable for software testing [2]. Practitioners in software developments always seek more effective software testing methods and tools to reduce the risks of failure and to decrease of the cost [3]. Especially, automated software testing tools are much-needed for assisting agile software development. However, manual testing still remains in practice. Automatic generation of tests from model-based formal specifications have been considered, such as Z [4], and Alloy language [5]. Despite many researchers' continuous efforts, this research area has many more issues to be addressed in practical automatic software testing than we had expected. First, construction of precise and rigorous formal specifications to accurately reflect the client's requirements is more difficult than researchers have claimed in the literature. Since formal specifications require to use mathematics, such as set theory, logics, and algebra, mathematical expressions may cause comprehension



gaps between the developers and clients. Second, formal specifications do not handle intermediate changes of user's requirements after a design approval. A flexible response to intermediate change is a big challenge in formal methods [6].

We proposed a GUI-aided approach to constructing formal specifications [7], and carried out case studies [8,9] with the intention of facilitating communications between software developers and the clients. The GUI-aided approach suggests how to construct formal specifications from informal requirements, and how to reduce intermediate changes of user's requirements by adapting a technique for rapid GUI prototyping. Rapid prototyping is widely used in software industry. A prototype in our proposed approach contributes to discovering functional behaviors, including end-users' implicit requirements [8]. Additionally, we used the Structured Object-Oriented Formal Language (SOFL) [10] as a formal specification language in order to define precisely and unambiguously required functions. SOFL is formed by an integration of VDM-SL [11], Data Flow Diagram [12], and Petri nets [13]. The functional behaviors specify as predicate expressions in SOFL at a series of state transitions at the system level, and translate a set of operations into the corresponding programs for producing output data from input data. In other words, the formal specifications comprise a set of the functional scenarios that require to be implemented as program paths in the corresponding programs.

We take advantage of the GUI-aided approach to build formal specifications with a comprehensible interface to the user for producing comprehensive test cases of the user's interest. The test case generation can be done using the test strategy proposed previously in our research [14]. This strategy adopts a disjunctive normal form approach [15] for deriving functional scenarios from the formal specifications. This paper presents a case study to develop a travel reservation system using both the GUI-aided approach for constructing formal specifications and the specification-based testing strategy for testing the corresponding program. In the case study, we generate test cases from a SOFL specification manually where each test case is composed of a test data and the expected result. Our experience has given us a chance to discover a challenge to the testing method that is how to verify external database accesses defined by a composite type with many attributes.

The rest of this paper is organized as follows. Section 2 introduces the GUI-aided approach, and explains a GUI model and formal specification in our case study. Section 3 mentions the basic concept of specification-based testing, and describes how to apply the specification-based testing in the case study. Section 4 discusses the relation between testing and prototyping. Lastly, in Sect. 5, we give conclusions and point out future work.

## 2 A GUI-Aided Approach

As we mention above, our GUI-aided approach intends to apply the advantages of rapid prototyping techniques for constructing formal specifications. Rapid prototyping techniques in software developments make it possible to improve

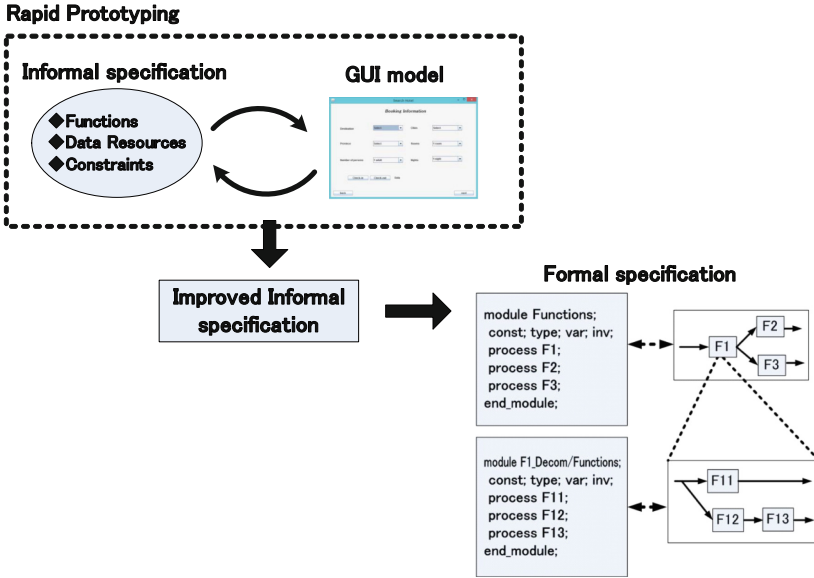
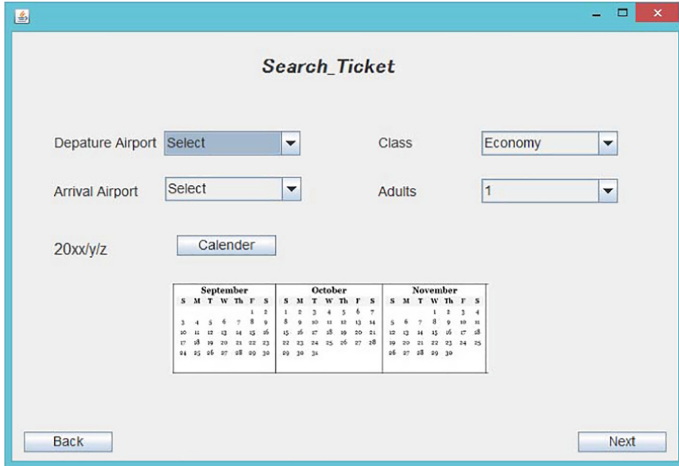


Fig. 1. A framework of a GUI aided Approach

communication between software developers and the clients, not to mention the fact that they reduce the development cost. Figure 1 illustrates a framework of the GUI-aided approach, which is composed of the three steps: rapid prototyping, improved informal specification, and formal specification.

1. A developer begins to define an informal specification which is written in natural language at the rapid prototyping step. Informal specifications consist of required functions, data resources, and constraints. Then, the developer implements a GUI model for the purpose of showing the potential behavior, discusses with the clients about desirable functions, and gets the clients' feedback for refining the model. What is more, this interaction between the developer and the client using the GUI model repeats until all necessary items identify on behalf of improving the initial informal specification.
2. The developer improves the initial informal specification using the GUI model. The second step demands to declare input data and output data, each parameter, and each value as necessary required functions. Therefore, we apply an animation technique by giving a demonstration to the client. Our GUI model incorporates buttons, menu bar, menu items, and event handlers. All event handlers are not necessarily associated with any database, but they represent system behaviors by a sequence of motions. The sequence of motions indicates a hierarchical structure of the required functions and the corresponding input and output data items with a button-related function.
3. Finally, the improved informal specification translates into a formal specification. In the formal specification, all of the functions are defined by a sequence

of modules, that are specified in a hierarchical fashion. Each module formally defines a functional abstraction. To be precise, a module has a module name, constant declarations, type declarations, variable declarations, an invariant section, and a list of process names. Each process is associated with conditional data flow diagram (CDFD). A CDFD represents an overall behavior and an association with a module. On the bottom right-hand side in Fig. 1, each small rectangle indicates an operation defined as a process in the related module, and each directed line represents a data flow.



**Fig. 2.** A GUI model for Search Ticket

For example, our target system required to serve some necessary functions for the purpose of finding reasonable flights from many airlines, travel agents and travel sites by searching, comparing, and booking online. The case study was started to write an informal specification based on this user requirement by one designer. And then, the designer implemented GUI models depicted in Fig. 2 with the aim of eliciting real requirement behind the apparent one. The models were implemented under the Eclipse environment using Swing in Java. Through the rapid prototyping step with eight research collaborators' feedback, the informal specification was improved for transforming into a formal specification.

Figure 3 represents a part of a module for the travel reservation system. The keyword **module** indicates the beginning of a module, and a module introduces the module name: `Search_Ticket_Decom`. The keyword **type** denotes type declarations to define all necessary data types for specifying related data resources. Each data resource is declared as a variable with an appropriate type. The type declaration of `BookingRequest` shows the composite type which has many attributes: departure airport, arrival airport, boarding date, seat class, and the number of seats. Since each attribute has appropriate type, the developer needs to consider the data structure respectively.

```

module Search_Ticket_Decom/Travel_System:
  type
    BookingRequest = composed of
      departure:Airport
      arrival:Airport
      date:Date
      class:{<Economy>, <Business>, <First>}
      request_seat_num:nat
    end;
    Airplane = composed of
      company:string
      flight_num:string
      departure:Airport
      arrival:Airport
      flight_schedule:Time
      seat_class:{<Economy>, <Business>, <First>}
      vacancy_inf:Vacancy
    end;
    .....
  ext plains: Airplane:
    .....

```

**Fig. 3.** The module of Search\_Ticket\_Decom

Modules specify a set of processes for the sake of providing functional operations. A process defines a process name, input and output ports, pre-condition, and post-condition. The pre-condition describes a constraint on the input data flows before the execution of the process, while the post-condition provides a constraint on the output data flows after the execution. SOFL requires to use simple propositional logics, basic set theory and predicates for describing formal specifications. For example, Fig. 4 describes the process `Search_Ticket` which defined by the functional operation of the GUI model depicted in Fig. 2.

A CDFD as depicted in Fig. 5 expresses two processes in the related module `Search_Ticket_Decom`: the process `Search_Ticket` for searching flight tickets based a request, and the process `Sort_Ticket` for sorting the search result. The process `Search_Ticket` takes one input data: `search_req`, accesses a database: `planes`, and produces three output data: `no_airline`, `no_seat`, and `flight_list`. A sequence of the motions comes from the button-related function based on the GUI model illustrated in Fig. 2.

### 3 Specification-Based Testing

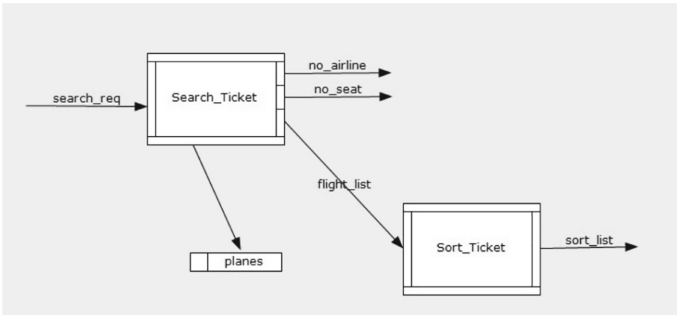
Model-based formal specification languages, such as VDM-SL [11], are typically described using propositional and predicate logics given in terms of pre- and post-conditions. Dick and Faivre [15] showed clearly to translate the pre- and post-conditions of a VDM specification into disjunctive normal form (DNF). We use this DNF approach for extracting the functional scenario which defines the

```

process Search_Ticket (search_req:BookingRequest)
    flight_list: Flight | no_airline: string | no_seat: string
ext wr planes
pre (exists[i: planes] | i.departure = search_req.departure and
    i.arrival = search_req.arrival)
post (exists[i: planes] | i.flight_schedule.date = search_req.date and
    i.seat_class.class = search_req.class and
    i.vacancy_inf.vacancy_num >= search_req.request_seat_num) and
    flight_list =conc(~flight_list, i) or
    not (exists[i: planes] | i.flight_schedule.date = search_req.date) and
    no_airline = "No service" or
    (exists[i: planes] | i.flight_schedule.date = search_req.date and
    not (exists[i: planes] | i.seat_class.class = search_req.class and
    i.vacancy_inf.vacancy_num >= search_req.request_seat_num) )and
    no_seat = "No vacancy"
end_process;

```

**Fig. 4.** The process of Search\_Ticket



**Fig. 5.** A CDFD for Search Ticket Decom

specific functional requirement or service in terms of getting input and generating output. A scenario can be specified by a predicate expression at the unit level or a series of data flow sequences at the system level in SOFL. Automatic derivations of functional scenarios from the pre- and post-conditions can be used parsers [16] for analyzing the syntax of the formal specification written in the SOFL specification.

Unfortunately, it is not easy to generate test cases for dealing with complex input expressions in the functional scenarios, including operations of compound data types, such as set, sequence, and composite type. More practical solutions for automatic generation of test cases needs to handle complex input expressions. Thus, we executed a case study for generation of test cases based on the functional scenarios involving operations of compound data types by manual. It had strong potential to provide a hint for automatic test case generation from the formal specification.

In this section, we will explain the test strategy based on DNF, how to derive test conditions from the operational specification of the target system, test cases (input-output pairs) in our case study and evaluation of the test results.

### 3.1 Test Strategy

Let  $S(S_{iv}; S_{ov})[S_{pre}; S_{post}]$  denote the specification of an operation  $S$ , where  $S_{iv}$  is the set of all input variables whose values are not changed by the operation,  $S_{ov}$  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-conditions of  $S$ , respectively.

#### Definition 1

Let  $S_{post} \equiv (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \dots \vee (C_n \wedge D_n)$ ,

where  $C_i$  ( $i = 1, \dots, n$ ) is the guard condition that contains no output variable in  $S_{ov}$ , and  $D_i$  is the defining condition that contains at least one output variable in  $S_{ov}$ . Then, a functional scenario  $f_s$  of  $S$  is a conjunction  $S_{pre} \wedge C_i \wedge D_i$ , and such an expression  $(S_{pre} \wedge C_1 \wedge D_1) \vee (S_{pre} \wedge C_2 \wedge D_2) \vee \dots \vee (S_{pre} \wedge C_n \wedge D_n)$  is called a functional scenario form (FSF). The conjunction  $S_{pre} \wedge C_i$  is called a testing condition of the scenario  $S_{pre} \wedge C_i \wedge D_i$ .

#### Definition 2

Let  $S_{iv} = \{x_1, x_2, \dots, x_r\}$  be the set of all input variables of operation  $S$  and  $Type(x_j)$  denotes the type of  $x_j$  ( $j = 1, \dots, r$ ). Then, a test case for  $S$ , denoted by  $T_c$ , is a mapping from  $S_{iv}$  to the set  $Values \cup \{nil\}$ :

$$T_c : S_{iv} \rightarrow Values \cup \{nil\}$$

$$T_c(x) \in Type(x) \vee T_c(x) = nil,$$

where  $Values = Type(x_1) \cup Type(x_2) \cup \dots \cup Type(x_r)$  and  $nil$  denotes the special value called “undefined”. With the above preparation, we define a test strategy.

#### Test strategy

Let operation  $S$  have an FSF  $(S_{pre} \wedge C_1 \wedge D_1) \vee (S_{pre} \wedge C_2 \wedge D_2) \vee \dots \vee (S_{pre} \wedge C_n \wedge D_n)$  where ( $n \geq 1$ ). Let  $T$  be a test set for  $S$ . Then,  $T$  must satisfy the condition  $(\forall_{i \in \{1, \dots, n\}} \exists_{t \in T} \cdot S_{pre}(t) \wedge C_i(t))$ . The symbol  $\forall$  which reads “for all” is called the universal quantifier, and the symbol  $\exists$  which reads “there exists” is called the existential quantifier. The condition  $(\forall_{i \in \{1, \dots, n\}} \exists_{t \in T} \cdot S_{pre}(t) \wedge C_i(t))$  means that for each functional scenario, there exist some test cases in test set  $T$  that satisfy its testing condition.

### 3.2 Test Conditions

We will explain how to generate test conditions by using the process Search\_Ticket illustrated in Fig. 4. Each variable of  $\{a, b, c, d, e\}$  denotes a propositional function, which has a matching value between two different data

**Table 1.** Test condition for the FSF

$a$	$b$	$c$	$d$	$e$	Expected output
T	T	T	T	T	f
T	T	T		F	h
T	T	T	F		h
T	T	F			g

resources in the process `Search_Ticket` or not. Variable  $f$  represents a sequence's operation, and variables  $g$  and  $h$  denote err messages.

$a$  : `i.departure = search_req.departure`

$b$  : `i.arrival = search_req.arrival`

$c$  : `i.flight_schedule.date = search_req.date`

$d$  : `i.seat_class.class = search_req.class`

$e$  : `i.vacancy_inf.vacancy_num >= search_req.request_seat_num`

$f$  : `flight_list = conc(~flight_list, i)`

$g$  : `no_airline = "No service"`

$h$  : `no_seat = "No vacancy"`

Then, a testing condition of the scenario  $S_{pre} \wedge C_i \wedge D_i$  derived from the  $S_{pre}$  and  $S_{post}$  in the process `Search_Ticket` based on our test strategy as follows:

$$(\exists(a \wedge b) \wedge \exists(c \wedge d \wedge e) \wedge f) \vee (\exists(a \wedge b) \wedge \neg \exists(c) \wedge g) \vee (\exists(a \wedge b) \wedge \exists(c) \wedge \neg \exists(d \wedge e) \wedge h)$$

The symbol  $\neg$  denotes negation of quantified statements. The conjunction can be applied by DeMorgan's law for generation of an FSF as follows:

$$\begin{aligned} & (\exists(a \wedge b) \wedge \exists(c \wedge d \wedge e) \wedge f) \vee (\exists(a \wedge b) \wedge \forall(\neg c) \wedge g) \vee (\exists(a \wedge b) \wedge \exists(c) \wedge \forall(\neg d \vee \neg e) \wedge h) \\ & \equiv (\exists(a \wedge b) \wedge \exists(c \wedge d \wedge e) \wedge f) \vee (\exists(a \wedge b) \wedge \forall(\neg c) \wedge g) \vee (\exists(a \wedge b) \wedge \exists(c) \wedge \\ & \forall(\neg d) \wedge h) \vee (\exists(a \wedge b) \wedge \exists(c) \wedge \forall(\neg e) \wedge h) \end{aligned}$$

The FSF contains four functional scenarios as follows:

1.  $(\exists(a \wedge b) \wedge \exists(c \wedge d \wedge e) \wedge f)$
2.  $(\exists(a \wedge b) \wedge \exists(c) \wedge \forall(\neg e) \wedge h)$
3.  $(\exists(a \wedge b) \wedge \exists(c) \wedge \forall(\neg d) \wedge h)$
4.  $(\exists(a \wedge b) \wedge \forall(\neg c) \wedge g)$

In summary, we have four test requirements based on the FSF, which can be satisfied with the values shown in Table 1. After the coding of the target system, we prepared test cases by manual using this test requirements based on the FSF derived from the formal specification.

### 3.3 Test Cases and Evaluation of the Test Results

In our case study, a programmer implemented the Java program with the intention of identifying the relationship between the formal specification and the corresponding program. The processes specified in SOFL were translated into a class or a set of classes in the program. The source code for our target system was implemented by 3.5 thousand of lines of code for about 33 hours. We executed a black-box testing, and found 9 defects: 3 defects of incorrect or missing functions, 5 defects of errors in data structures or external database access, and 1 defect of interface missing or erroneous as Table 2.

Table 3 represents the test case for the purpose of checking the source code related with the process `Search_Ticket` illustrated in Fig. 4. Besides, both of the input values and the expected output values in the test cases are originated from the test condition described in Table 1. Meanwhile, Table 4 shows the test condition, test result, and classification of defects. By way of illustration, we show the test case of No.2 in the Tables 3 and 4. In the test case, the number of seats requested for a flight reservation exceeded the seat availability under the condition of stored data in the external database. Despite less seat availability, the system made a reservation as possible number of seats without providing a detailed status. Not only that, it did not give any error message of shortage. This incorrect operation was caused by an inappropriate manipulation in composite data types.

Besides, we found fifth defects related data structures or external database access. Some of these defects forced termination of system, when the source code reads the data in the external database based on a request for booking flight. These defects obviously have a relationship with the external databases and the exception handlings associated with database accesses. The database related

**Table 2.** Test result

Classification of defects	Number of defects
Incorrect or missing functions	3
Data structures or external database access	5
Interface missing or erroneous	1

**Table 3.** Test cases for the FSF

Departure	Arrival	Date	Class	Seats	Expected output
HND	CTS	2017/12/21	Economy	1	add in flight_list
HND	CTS	2017/12/21	Economy	4	“No vacancy”
HND	CTS	2017/12/21	Business	1	“No vacancy”
HND	CTS	Null	Economy	1	“No service”
HND	NRT	2017/12/21	Economy	1	error message
⋮	⋮	⋮	⋮	⋮	⋮



**Table 4.** Test analysis

Departure & Arrival	Date	Class	Seat	Result	Classification of defects
T	T	T	T	Success	
T	T	T	F	Failed	Incorrect or missing functions
T	T	F	F	Success	
T	F	T	T	Failed	Interface missing or erroneous
F	T	T	T	Success	
⋮	⋮	⋮	⋮	⋮	⋮

operations are not included in our GUI model. The model incorporates buttons, menu bar, menu items, and event handlers, although the event handlers have no association with any database.

## 4 Discussions

Prototypes are often used in usability testing, because they allow for interactions between users and a product to be measured under controlled conditions. Usability testing is a technique for evaluating the effectiveness, efficiency, and user satisfaction in the software product or web service. Sauer and Sonderegger [17] reviewed of the nine research literature that the majority of studies concluded that few differences between computer and paper media or low and high fidelities in usability testing.

Just a few of researches mentioned test case generation based on prototypes and formal specifications. Treharne et al. [18] focused on a testing process based on the use of a prototype. In this research, B-Method is used to define a formal specification written in Abstract Machine Notation. Unfortunately, the research does not expand possibilities for automatic test case generation.

Many research papers explain automatic test case generation based on UML activity diagrams [19,20], or UML state chart diagrams [21]. However the UML diagrams are neither a prototype nor formal language, the test case generation algorithm may be useful for our future research.

## 5 Conclusions

This paper described an integrated methodology for constructing formal specifications by a GUI-aided approach and verifying programs by a specification-based testing strategy. Then, we applied the GUI-aided approach for refining a formal specification of a travel reservation system, implemented a program based on the formal specification, and checked the program by specification-based testing using functional scenarios derived from the formal specification. The GUI-aided

approach aims to find out desirable functions in the early phase of development through communications between software developers and the clients. The specification-based testing strategy uses the advantage of formal methods for verifying the correctness of programs, and it has a high potential for automatic generations of tests.

In the purpose of assessing automatic test case generation from the formal specification constructed by a GUI-aided approach, we executed a black-box testing. The result shows that we need to consider carefully the generation of test cases about the external databases and the exception handlings associated with database accesses. We will investigate how to generate test cases for database related operations, and exception handlings associated with database accesses from the formal specifications as our future work.

**Acknowledgement.** We would like to thank Daisuke Noguchi for developing GUI models, including writing the SOFL specifications and completing the implementation in Java. This work was supported by JSPS KAKENHI Grant Number 26240008.

## References

1. Dijkstra, E.W.: Structured Programming, pp. 1–82. Academic Press Ltd., London (1972)
2. Ammann, P., Offutt, J.: Introduction to Software Testing, 1st edn. Cambridge University Press, New York (2008)
3. Naik, K., Tripathy, P.: Software Testing and Quality Assurance: Theory and Practice, 1st edn. Wiley-Spektrum, Hoboken (2008)
4. Ammann, P., Offutt, J.: Using formal methods to derive test frames in category-partition testing. In: Proceedings of the Ninth Annual Conference on Computer Assurance, 1994. COMPASS 1994 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security, pp. 69–79 (1994)
5. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2000, pp. 14–25. ACM, New York (2000)
6. Bustard, D.W., Winstanley, A.C.: Making changes to formal specifications: requirements and an example. IEEE Trans. Softw. Eng. **20**, 562–568 (1994)
7. Liu, S.: A GUI-aided approach to formal specification construction. In: Liu, S., Duan, Z. (eds.) SOFL+MSVL 2015. LNCS, vol. 9559, pp. 44–56. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-31220-0\\_4](https://doi.org/10.1007/978-3-319-31220-0_4)
8. Nagoya, F., Liu, S.: A case study of a GUI-aided approach to constructing formal specifications. In: Liu, S., Duan, Z., Tian, C., Nagoya, F. (eds.) SOFL+MSVL 2016. LNCS, vol. 10189, pp. 74–84. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-57708-1\\_5](https://doi.org/10.1007/978-3-319-57708-1_5)
9. Nagoya, F., Liu, S.: A comparative study of a GUI-aided formal specification construction approach. In: Gervasi, O., et al. (eds.) ICCSA 2017. LNCS, vol. 10404, pp. 273–283. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-62392-4\\_20](https://doi.org/10.1007/978-3-319-62392-4_20)
10. Liu, S.: Formal Engineering for Industrial Software Development. Springer, Heidelberg (2004). <https://doi.org/10.1007/978-3-662-07287-5>
11. Jones, C.B.: Systematic Software Development Using VDM. Prentice Hall International (UK) Ltd., Manchester (1986)

12. DeMarco, T.: *Structured Analysis and System Specification*. Prentice Hall PTR, Upper Saddle River (1979)
13. Reisig, W.: *Petri Nets: An Introduction*. Springer, New York (1985). <https://doi.org/10.1007/978-3-642-69968-9>
14. Liu, S., Nakajima, S.: A decompositional approach to automatic test case generation based on formal specifications. In: *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. SSIRI 2010*, pp. 147–155. IEEE Computer Society, Washington (2010)
15. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Woodcock, J.C.P., Larsen, P.G. (eds.) *FME 1993. LNCS*, vol. 670, pp. 268–284. Springer, Heidelberg (1993). <https://doi.org/10.1007/BFb0024651>
16. Liu, S.: A tool supported testing method for reducing cost and improving quality. In: *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Vienna, Austria, 1–3 August 2016*, pp. 448–455. IEEE (2016)
17. Sauer, J., Sonderegger, A.: The influence of prototype fidelity and aesthetics of design in usability tests: effects on user behaviour, subjective evaluation and emotion. *Appl. Ergon.* **40**, 670–677 (2009)
18. Treharne, H., Draper, J., Schneider, S.: Test case preparation using a prototype. In: Bert, D. (ed.) *B 1998. LNCS*, vol. 1393, pp. 293–311. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0053368>
19. Mingsong, C., Xiaokang, Q., Xuandong, L.: Automatic test case generation for UML activity diagrams. In: *Proceedings of the 2006 International Workshop on Automation of Software Test. AST 2006*, pp. 2–8. ACM, New York (2006)
20. Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., Guoliang, Z.: Generating test cases from UML activity diagram based on gray-box method. In: *Proceedings of the 11th Asia-Pacific Software Engineering Conference. APSEC 2004*, pp. 284–291. IEEE Computer Society, Washington (2004)
21. Ali, S., Briand, L.C., Rehman, M.J.U., Asghar, H., Iqbal, M.Z.Z., Nadeem, A.: A state-based approach to integration testing based on UML models. *Inf. Softw. Technol.* **49**, 1087–1106 (2007)

# **Graph Theory**



# On the Cooperative Graph Searching Problem

Chin-Fu Lin, Ondřej Navrátil, and Sheng-Lung Peng<sup>(✉)</sup>

National Dong Hwa University, Hualien, Taiwan  
{810521001,810321005,slpeng}@mail.ndhu.edu.tw

**Abstract.** In this paper, we introduce a new variation of graph searching problem, namely, cooperative graph searching problem. We define that a searcher is isolated if there is no other searchers on its close neighborhood. In this variant, we add an additional constrain that every searcher would not be isolated after each searching step. Therefore, we can make sure that every searcher can be cooperated by another searcher. We prove that the cooperative graph searching problem is NP-complete on general graphs and propose polynomial-time algorithms for the problem on grid graphs.

**Keywords:** Graph searching · Mixed searching  
Cooperative graph searching

## 1 Introduction

The *graph searching problem* was first proposed by Breisch [1]. Parsons contributed some earlier works for the problem [2,3]. In the problem, a graph  $G$  represents a system of tunnels. Initially, all the edges of  $G$  are contaminated by a gas. An edge is *cleared* by some operations on  $G$ . A cleared edge is *recontaminated* if there is a path from an uncleared edge to the cleared edge without any searchers. The objective is to use as few searchers as possible to make all edges be cleared. The allowable operations are as follows:

1. Place a searcher on a vertex.
2. Remove a searcher from a vertex.
3. Move a searcher along an edge.

If an edge is only cleared by moving a searcher along the edge, then it is called the *edge searching problem*. On the other hand, if an edge is cleared by having two searchers on both its two ends, then it is called the *node searching problem*. If both of clearing rules are allowed, then it is called the *mixed searching problem*. The mixed searching problem was first proposed by Takahashi et al. [4]. It is obvious that the mixed searching problem is a natural generalization of edge and node searching problems. The mixed search number of  $G$ , denoted by  $ms(G)$ , is the minimum number  $k$  such that  $G$  is  $k$ -searchable by mixed search rules.

These variants of graph searching problems are not only interesting theoretically, but also have applications on problems like embedding tree queries [5,6], key graph searching [7,8], selectivity estimation [9], subgraph matching [10,11], min cut finding [12], and so on.

Since graph searching problems are so powerful, there are many variants of searching game for finding more appropriate strategies proposed for modeling real world problems. Blin proposed a searching problem named *exclusive graph searching problem* [13]. Latter, Markou studied a monotone version of exclusive graph searching [14]. Exclusive graph searching is a variant of mixed searching with the following extra constrains:

1. Initially, place all the searchers in the graph.
2. Every vertex can contain only one searcher.

Dyer introduced the fast searching problem based on edge searching in which we can traverse each edge only once [15]. Xue proposed algorithms for some special graphs [16]. Even though there are so many variants of searching game, most of them put emphasis on efficiency. However, in the real world, we should care not only performance but also searchers' safety. A searcher is *isolated* if there is no another searcher on its neighbor vertex or on the same vertex. To avoid a secondary distress/injure on rescue works and polices' raids, each searcher has better not to be isolated.

For guaranteeing that every searcher will not be isolated, we define a variant of searching game, called the *cooperative graph searching problem*. Initially, we have a graph  $G = (V, E)$  in which every edge  $e \in E$  is contaminated. Our target is to clean all the edges and make sure all searchers are under cooperative for each searching step. The clearing rules are based on mixed searching. However, for making a possible cooperation between searchers. we have the following possible operations.

1. Place a searcher on a vertex.
2. Place two searchers on the end-vertices of an edge.
3. Remove a searcher from a vertex.
4. Remove two searchers from the end-vertices of an edge.
5. Move a searcher along an edge.

For each step, only one operation above is allowed. We use  $S_i = (E_i, C_i)$  to denote the status after operation  $i$  is applied, where  $E_i$  (respectively,  $C_i$ ) denotes the set of uncleared (respectively, cleared) edges. A sequence of  $S_0, S_1, \dots, S_r$  that clears the graph  $G$  is called a search strategy of  $G$  and is denoted as  $\mathcal{S}$ . By the definition,  $S_0 = (E, \emptyset)$  and  $S_r = (\emptyset, E)$ . Note that  $E_i \cap C_i = \emptyset$  and  $E_i \cup C_i = E$  for each step  $i$ . Let  $|S_i|$  denote the number of searchers on graph after step  $i$ . Let  $|\mathcal{S}| = \max_i |S_i|$  denote the number of searchers used to clear  $G$ . A search strategy is *cooperative* if for each step there is a searcher on some vertex, then there is another searcher at its neighbor vertex or stay at the same vertex. A cooperative search strategy is *optimal* if it uses the minimum number of searchers to clear  $G$ . The cooperative graph searching problem on  $G$  is to find an optimal cooperative search strategy to clear  $G$ .

We say a graph  $G$  is  $k$ -searchable if we can clear  $G$  by using at most  $k$  searchers. Similarly,  $G$  is  $k$ -cooperative searchable if  $G$  can be cleared by using at most  $k$  searchers using a  $k$ -cooperative search strategy. The cooperative search number of  $G$ , denoted by  $\text{cos}(G)$ , is the minimum number  $k$  such that  $G$  is  $k$ -cooperative searchable. Thus the decision version of the cooperative graph searching problem is called the  $k$ -cooperative graph searching problem that asks whether  $G$  is  $k$ -searchable or not?

## 2 Preliminary Results

Let  $G = (V, E)$  be a simple, finite, and undirected graph. For a vertex subset  $W \subseteq V$ , let  $G[W]$  be a subgraph of  $G$  that is induced by  $W$ . A *clique* is a complete subgraph and an *independent set* is a subgraph that has no edge. Let  $K_n$  (respectively,  $P_n$  and  $C_n$ ) be a complete (respectively, path and cycle) graph with  $n$  vertices. The following lemma shows some easy results.

**Lemma 1.** *The following statements are true.*

1.  $\text{cos}(K_n) = n - 1$  for  $n \geq 3$ .
2.  $\text{cos}(P_n) = 2$  for  $n \geq 2$ .
3.  $\text{cos}(C_n) = 4$  for  $n \geq 5$ .

*Proof.* For  $K_n$ , it is not hard to check that  $n - 2$  searchers are not enough to clear the graph. Since after the  $n - 2$  searchers are on the graph, there are two remaining vertices that cannot be guarded and therefore no way can clear the edge between them without recontamination. However, we can clear  $K_n$  by the following strategy.

1. Firstly, place 2 searchers on the end-vertices of an edge.
2. Then, place the remaining  $n - 3$  searchers on any unguarded vertices one by one.
3. Finally, move any one searcher to the remaining unguarded vertex.

Since  $K_n$  is a complete graph, all the searchers on vertices are adjacent. Thus, it is a cooperative search strategy.

For  $P_n$ , let  $P_n = (v_1, v_2, \dots, v_n)$ . To clear  $P_n$ , we first place two searchers on  $v_1$  and  $v_2$ . Then, move the searcher at  $v_1$  to  $v_2$  and then move a searcher on  $v_2$  to  $v_3$ . The following steps are similar. That is, these two searchers cooperatively search the path until to  $v_n$ . Thus  $\text{cos}(P_n) = 2$ .

For  $C_n$ , The idea is similar to the search strategy of  $P_n$ . However, to avoid recontamination, we need two team from different directions to clear the graph. By the cooperative rules, the minimum number of each team is 2. Thus  $\text{cos}(C_n) = 4$  for  $n \geq 5$ .  $\square$

**Lemma 2.** *Let  $G'$  be an induced subgraph of  $G$ . Then  $\text{cos}(G') \leq \text{cos}(G)$ .*

*Proof.* Consider a cooperative search strategy  $\mathcal{S}$  of no recontamination for clearing  $G$ . Since  $\mathcal{S}$  has no recontamination, all the searchers on a vertex can be moved/removed only when the vertex is clear, *i.e.*, no any contaminated incident edge. In  $\mathcal{S}$ , we keep each step for cleaning edges in  $G'$ . We delete those steps that clean edges which are not in  $G'$ . Thus,  $G'$  can be cleared using the modified strategy. Clearly,  $\text{cos}(G') \leq \text{cos}(G)$ .  $\square$

**Theorem 1.** *For any graph  $G = (V, E)$ ,  $ms(G) \leq \text{cos}(G) \leq 2ms(G)$  and the bound is tight.*

*Proof.* Since cooperative graph searching problem is a special case of mixed searching problem, the lower bound  $ms(G) \leq \text{cos}(G)$  is obvious. Now consider an optimal mixed search strategy  $\mathcal{S}$  for  $G$ . In  $\mathcal{S}$ , for each step if it violates the cooperative search rules, then we add an extra searcher to a neighbor of the target vertex which will violate the rules. For example, if we want to clear the edge  $uv$  by moving a searcher from  $u$  to  $v$ , then there are the following two cases that the cooperative rules will be violated.

1. the new coming searcher at  $v$  is isolated.
2. the missing searcher of  $u$  causes an isolated searcher on a neighbor of  $u$ .

For both cases, we can just place an extra searcher to  $v$  such that the edge  $uv$  is cleared by both of its two ends having a searcher. Thus our upper bound is obtained.

Finally, by Lemma 1, for  $n \geq 3$   $ms(K_n) = \text{cos}(K_n) = n - 1$  which meets the lower bound. Similarly, for  $n \geq 4$   $\text{cos}(C_n) = 2ms(C_n) = 4$  which meets the upper bound. Thus the bounds are tight.  $\square$

**Corollary 1.** *For any tree  $T$ ,  $\text{cos}(T) \leq 2ms(T) = O(\lg n)$ .*

**Lemma 3.** *Let  $G = (V, E)$  be a graph containing two cliques  $K_m$  and  $K_n$  such that  $V = K_m \cup K_n$  and  $m \geq n \geq |K_m \cap K_n|$ . Then,  $\text{cos}(G) = m - 1$ .*

*Proof.* Since  $K_m \subseteq G$ , by Lemmas 1 and 2,  $\text{cos}(G) \geq m - 1$ . To show that  $\text{cos}(G) = m - 1$ , we propose the following strategy to clear  $G$  by using  $m - 1$  searchers.

1. Let  $u$  be in  $K_m \cap K_n$  and  $v$  be in  $K_n \setminus K_m$ .
2. Place  $m - 1$  searchers on  $K_m \setminus \{u\}$ .
3. Move one searcher in  $K_m \setminus K_n$  to  $u$ .
4. Remove all the searchers from  $K_m \setminus K_n$ .
5. Place searchers on  $K_n \setminus (K_m \cup \{v\})$ .
6. Move any searcher on  $G$  to  $v$ .

For the above strategy, it is easy to check that we use at most  $m - 1$  searchers to clear  $G$  if  $m \geq n$ . It proves the lemma.  $\square$

A graph  $G = (V, E)$  is a split graph if  $V$  can be partitioned into two sets  $C$  and  $S$  such that the induced subgraph  $G[C]$  is a clique and  $G[S]$  is an independent set. For convenience, we use  $G = (C \cup S, E)$  to denote a split graph.



**Theorem 2.** *Let  $G = (C \cup S, E)$  be a split graph. Then  $|C| - 1 \leq \text{cos}(G) \leq |C| + 1$ .*

*Proof.* Since  $G[C]$  is a clique of  $G$ , by Lemmas 1 and 2, we obtain that  $|C| - 1$  is a lower bound of  $\text{cos}(G)$ . On the other hand, we can clear  $G$  by using  $|C| + 1$  searchers as follows.

1. Place  $|C|$  searchers on vertices of  $C$ .
2. Place one extra searcher on  $S$  one after one.

It is not hard to check that the above strategy can clear  $G$  using  $|C| + 1$  searchers. Therefore, we have this theorem.  $\square$

### 3 NP-Completeness Result

A search strategy can be recontaminated if there is an edge which is cleared at step  $i$  but becoming unclear at step  $j$  for  $j > i$ . However, for edge searching, LaPaugh showed the following theorem.

**Theorem 3** ([17]). *If graph  $G$  is  $k$ -searchable, then there is a search strategy using at most  $k$  searchers without recontamination.*

By using a similar argument as the proof of Theorem 3, we can show the following theorem. We omit the proof for this version.

**Theorem 4.** *If graph  $G$  is  $k$ -cooperative searchable, then there is a cooperative search strategy using at most  $k$  searchers without recontamination.*

By Theorem 4, we may assume that the cooperative search strategy we considered does not be recontamination.

**Lemma 4.** *The  $k$ -cooperative graph searching problem is in NP.*

*Proof.* For any graph  $G = (V, E)$  and a given integer  $k$ , we design a non-deterministic polynomial-time algorithm that checks whether  $G$  is  $k$ -cooperative searchable or not. The detailed algorithm is shown in Algorithm 1. It shows the lemma.  $\square$

**Theorem 5** ([18]). *The mixed searching problem is NP-Complete.*

**Theorem 6.** *The  $k$ -cooperative graph searching problem is NP-Complete.*

*Proof.* By Lemma 4, the  $k$ -cooperative graph searching problem is in NP. To complete the proof, the remaining work is to show it is NP-hard. Our reduction is from the mixed searching problem.

For any graph  $G = (V, E)$ , we construct an extended graph  $G' = (V', E')$  by adding a universal vertex  $u$ . That is,  $V' = V \cup \{u\}$  and  $E' = E \cup \{uv \mid v \in V\}$ . We claim that  $G$  can be mixed searched using at most  $k - 1$  searchers if and

**Algorithm:**  $k$ -cooperative searchable

**Data:** a graph  $G = (V, E)$  and an integer  $k$

**Result:**  $G$  is  $k$ -cooperative searchable or not.

$F \leftarrow E$ ;

**while**  $F$  is not empty **do**

    nondeterministically, select an uncleared edge  $e = uv \in F$

**if** both end-vertices of  $e$  do not have a searcher **then**

        | place two searchers on the end-vertices of  $e$

**else**

        assume a searcher is at  $u$ ;

**if** all the edges incident to  $u$  except  $uv$  are clear **then**

            | move the searcher at  $u$  to  $v$

**else**

            | place a searcher on  $v$

**end**

**end**

$F \leftarrow F \setminus \{e\}$ ;

**if** the number of searchers used is greater than  $k$  **then**

        | return *False*

**else**

        Remove searchers if they are not needed under the constraints of cooperative search rules;

**end**

**end**

return *True*;

**Algorithm 1.** NP algorithm for checking  $k$ -cooperative searchable.

only if  $G'$  can be cooperatively searched by at most  $k$  searchers. The only if part is easy. In mixed searching, if  $v$  is the first vertex that is guarded by a searcher, then we simultaneously place another searcher on vertex  $u$ . Thus, they are cooperative since for each searcher on a vertex of  $V$  there is always a searcher on  $u$ . Then following the mixed searching strategy on  $G$  we can clear  $G'$  obeying the cooperative searching rules.

On the other hand, if we can cooperatively search  $G'$  by using at most  $k$  searchers, then  $G$  can be mixed searched with  $k-1$  searchers. Consider a cooperative search strategy  $\mathcal{S}$  that does not contain recontaminated edges by Theorem 4. Since  $u$  does not exist in  $G$ , we remove every step that clears an edge incident to  $u$ . Let the resulting strategy be  $\mathcal{S}'$ . By definition  $\mathcal{S}'$  is a mixed search strategy that clears  $G$ . Since  $|\mathcal{S}| = k$  and  $u$  is a universal vertex in  $G'$ , when there are  $k$  searchers on  $G'$ ,  $u$  must be guarded by one. Otherwise, we need extra searcher to guard  $u$  for maintaining recontamination. Therefore  $|\mathcal{S}'| = k-1$ . This proves the theorem.  $\square$

## 4 Polynomial-Time Algorithm for Grid Graphs

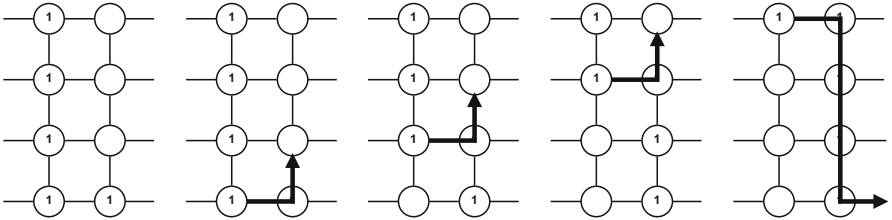
*Grid graphs* are a class of graphs that is the graph Cartesian product of path graphs. Two dimensional grid graphs are the most representative case of grid

graphs and have many applications. For example, we can simulate roads network of modern planned cities by grid graphs, or simulate electronic circuits by grid graphs with graph embedding skill. By definition, we can represent any two dimensional grid graphs by a production of two paths  $P_m$  and  $P_n$ . We denote it by  $G_{m \times n}$  assuming that  $m \geq n$ .

**Theorem 7.** For any grid graph  $G_{m \times n}$ ,  $n \geq 3$ ,  $\text{cos}(G_{m \times n}) = n + 1$ .

*Proof.* We assume that in  $G_{m \times n}$  there are  $m$  columns and  $n$  rows. We propose the following strategy to clear  $G_{m \times n}$ . It works column by column. We first clear the first column by placing  $n$  searchers on vertices of it. Suppose that the  $i$ -th column is cleared. We are going to clear the  $(i + 1)$ -th column. For clearing edges between  $i$ -th column and  $(i + 1)$ -th column, and obeying cooperative rules, we need an extra searcher, namely,  $(n + 1)$ -th searcher to complete the work. The clearing progress is shown in Fig. 1. Thus  $\text{cos}(G_{m \times n}) \leq n + 1$ . By applying a separator theorem [19] on  $G_{m \times n}$ ,  $\text{cos}(G_{m \times n}) \geq n$ .

Assume that we want to clear the graph by using only  $n$  searchers. To clear edge between two columns, our searchers have to guard vertices in these two columns. Since we can only move one searcher in an operation, the first searcher moving (or placing) to  $(i + 1)$ -th column have to have a neighbor searcher at the same row. Otherwise, it is not cooperative. Note that at this moment, all the  $n$  searchers on column  $i$  cannot be moved (or removed); otherwise, a recontamination occurs. Thus  $n$  searchers are not enough to clear the graph  $G_{m \times n}$ . Hence,  $n < \text{cos}(G_{m \times n}) \leq n + 1$ , i.e.,  $\text{cos}(G_{m \times n}) = n + 1$ .  $\square$



**Fig. 1.** A progress for clearing  $G_{m \times n}$ .

Grid graphs are an important class of interconnection networks. For a more general model, some vertices may be failure. That is, we can remove these failed vertices from  $G$ . To search such a grid, we use the same approach (column by column) mentioned above to clear it. Assume that all the vertices of the  $i$ -th column are good. Thus when we want to march to column  $i + 1$ , vertices on column  $i + 1$  are either good or bad. It divides the column into a set of paths. In particular, some paths contain only one vertex. Thus for cooperative rules, we need a searcher stand at the vertex on column  $i$  for supporting the searcher marching to column  $i + 1$ . For those paths of length at least two, we use the same

technique to go to column  $i + 1$ . Assume that we have  $r$   $P_1$ 's in column  $i + 1$ . By cooperative rules, we need  $r$  supporting vertices guarded on column  $i$ . The remaining problem is that do we have enough searchers to do the job? In the worst case, we have at least  $r$  missing vertices on column  $i + 1$ . Thus,  $n$  searchers are sufficient to guard vertices on column  $i + 1$  and supporting vertices on column  $i$ . However, we still need one extra searcher to help searchers on column  $i$  to column  $i + 1$ . Therefore, we need  $n + 1$  searchers for clearing this kind of grid graphs. For the other special cases, the arguments are similar. We omit the detail in this version. Finally, we have the following theorem.

**Theorem 8.** *Grid graphs  $G_{m \times n}$  without some vertices are  $(n + 1)$ -cooperative searchable.*

## 5 Conclusion

In this paper, we propose the cooperative graph searching problem, a new variant of graph searching problem by including cooperative rules. The rules make sure that no searcher will be alone or isolated during a searching process. We believe it is more suitable to model some real world problems. We propose some properties on graphs for this problem. In particular, we show that this problem is NP-complete on general graphs and it can be solved on grid graphs. In [16], the author solved the exclusive graph searching problem in polynomial time. We believe that by a similar idea our algorithm can be generalized for generalized grid graphs, *i.e.*,  $k$  dimensional grid graphs for any  $k$ . In the future, we want to study features of cooperative graph searching problem on other graph classes, *i.e.*, trees. We believe that this problem on trees can be solved in polynomial time.

**Acknowledgement.** This work was partially supported by the Ministry of Science and Technology of Taiwan, under Contract No. MOST 105-2221-E-259-018.

## References

1. Breisch, R.: An intuitive approach to speleotopology. *Southwest. Cavers* **6**(5), 72–79 (1967)
2. Parsons, T.: Pursuit-evasion in a graph. In: Alavi, Y., Lick, D.R. (eds.) *Theory and Applications of Graphs*, pp. 426–441. Springer, Heidelberg (1976). <https://doi.org/10.1007/BFb0070400>
3. Parsons, T.: The search number of a connected graph. In: *Proceedings of the 9th Southeastern Conference on Combinatorics, Graph Theory and Computing*, pp. 549–554 (1978)
4. Takahashi, A., Ueno, S., Kajitani, Y.: Mixed searching and proper-path-width. *Theor. Comput. Sci.* **137**, 253–268 (1995)
5. Cole, R., Hariharan, R., Indyk, P.: Tree pattern matching and subset matching in deterministic  $O(n \log^3 n)$ -time. In: *Proceedings of the ACM-SIAM Symposium on Discrete Algorithm*, pp. 245–254 (1999)

6. Hoffman, C., O'Donnell, J.: Pattern matching in trees. *J. ACM* **29**, 68–95 (1982)
7. Cook, D., Holder, L.: Substructure discovery using minimum description length and background knowledge. *J. Artif. Intell. Res.* **1**, 231–255 (1993)
8. Djoko, S., Cook, D., Holder, L.: An empirical study of domain knowledge and its benefits to substructure discovery. *IEEE Trans. Knowl. Data Eng.* **9**, 575–586 (1997)
9. Chen, Z., Korn, F., Koudas, N., Muthukrishnan, S.: Selectivity estimation for Boolean queries. In: *Proceedings of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pp. 216–225 (2000)
10. Almohamad, H., Duffuaa, S.: A linear programming approach for the weighted graph matching problem. *IEEE Trans. Pattern Anal. Mach. Intell.* **15**, 522–525 (1993)
11. Christmas, W., Kittler, J., Petrou, M.: Structural matching in computer vision using probabilistic relaxation. *IEEE Trans. Pattern Anal. Mach. Intell.* **17**, 749–764 (1995)
12. Chung, M., Makedon, F., Sudborough, I., Turner, J.: Polynomial time algorithm for the min cut problem on degree restricted trees. *SAIM J. Comput.* **14**, 158–177 (1985)
13. Blin, L., Burman, J., Nisse, N.: Exclusive graph searching. In: Bodlaender, H.L., Italiano, G.F. (eds.) *ESA 2013*. LNCS, vol. 8125, pp. 181–192. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40450-4\\_16](https://doi.org/10.1007/978-3-642-40450-4_16)
14. Markou, E., Nisse, N., Pérennes, S.: Exclusive graph searching vs pathwidth. *Inf. Comput.* **252**, 243–260 (2017)
15. Dyer, D., Yang, B., Yaşar, Ö.: On the fast searching problem. In: Fleischer, R., Xu, J. (eds.) *AAIM 2008*. LNCS, vol. 5034, pp. 143–154. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-68880-8\\_15](https://doi.org/10.1007/978-3-540-68880-8_15)
16. Xue, Y., Yang, B.: The fast search number of a Cartesian product of graphs. *Discrete Appl. Math.* **224**, 106–119 (2017)
17. LaPaugh, A.: Recontamination does not help to search a graph. *J. ACM* **40**, 224–245 (1993)
18. Yang, B., Cao, Y.: Monotonicity in digraph search problems. *Theor. Comput. Sci.* **407**, 523–544 (2008)
19. Escalante, F.: Schnittverbände in graphen. *Abh. Math. Semin. Univ. Hambg.* **38**, 199–220 (1972)

# **Model Checking**



# Boosting UPPAAL for OSEK/VDX Applications with a Sequentialization Approach

Haitao Zhang<sup>1</sup>(✉), Zhuo Cheng<sup>2</sup>, Jianxin Xue<sup>3</sup>, and Yonggang Lu<sup>1</sup>

<sup>1</sup> School of Information Sciences and Engineering, Lanzhou University,  
Lanzhou 730000, China  
{htzhang,ylu}@lzu.edu.cn

<sup>2</sup> State International S&T Cooperation Base of Networked Supporting Software,  
Jiangxi Normal University, Nanchang 330022, China  
zhuo\_cheng@126.com

<sup>3</sup> School of Computer and Information Engineering,  
Shanghai Polytechnic University, Shanghai 201209, China  
jxxue@sspu.edu.cn

**Abstract.** The OSEK/VDX standard has been widely adopted by automotive manufacturers for vehicle mounted systems. The ever increasing complexity of the system has created a challenge for examining the timing properties of the developed OSEK/VDX applications in exhaustive way, such as reachability property. Model checking as an exhaustive verification technique has attracted great attentions in the automotive industry. To verify OSEK/VDX applications by using model checking, a tentative method has been proposed based on the model checker UPPAAL. However, the existing method is usually not scalable to verify a large-scale OSEK/VDX application since the constructed application model is too complex. In this paper, we propose an efficient approach to simplify the application model for making UPPAAL more scalable in verifying large-scale OSEK/VDX applications. We evaluated our approach based on a series of experiments. The experimental results show that our approach is not only capable of efficiently simplifying the OSEK/VDX application models, but also of making the model checker UPPAAL competent in dealing with the OSEK/VDX applications with industrial complexity.

## 1 Introduction

With the development of automotive industry and electronic technology, more and more complex vehicle-mounted systems are deployed in vehicles. However, how to reuse and transplant the developed systems has become a serious problem for the automotive manufacturers, since there is no uniform development standard in the automotive industry. To finish off this problem, European Automobile Manufacturer Association develops and promulgates a vehicle-mounted system standard named OSEK/VDX [15] in 1994. The standard has now been widely adopted by many automotive manufacturers and research groups to implement a

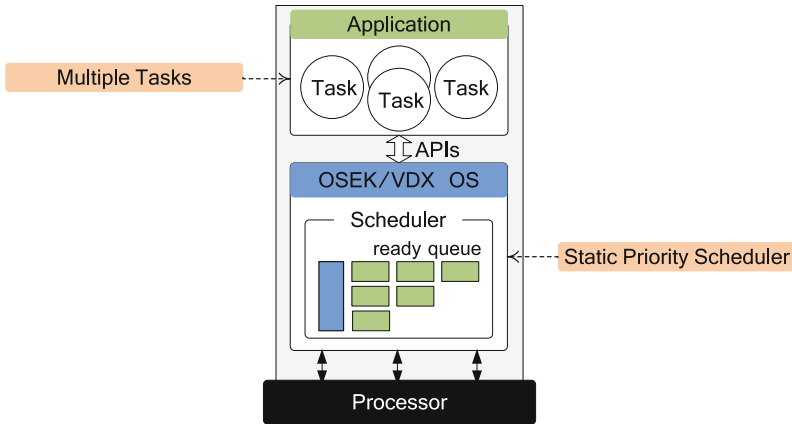


Fig. 1. Structure of the OSEK/VDX vehicle-mounted system.

practical and study vehicle-mounted system, e.g., the automotive manufacturers BMW, Audi, Volkswagen, and the research groups IRCCyN, TOPPERS, etc.

In general, a developed OSEK/VDX vehicle-mounted system includes two important components, one is operating system (OS), and the other one is applications. Currently, alongside the improvement of vehicle automation, more and more OSEK/VDX applications are developed for assisting drivers based on the OSEK/VDX OS. As shown in Fig. 1, a developed OSEK/VDX application consists of multiple tasks. When the application is running on the OSEK/VDX OS, tasks within the application are concurrently executed under the scheduling of the OSEK/VDX OS, where a *deterministic* scheduler called static priority scheduler is adopted by the OSEK/VDX OS to dispatch tasks, especially a ready queue is used to manage the scheduling order of tasks. In addition, tasks can invoke application interfaces (APIs) to dynamically change the scheduling order of tasks, e.g., activate a higher priority task. As a result of the concurrency of tasks and dynamic of scheduling, developers face the challenge of exhaustively ensuring the timing properties of the developed OSEK/VDX applications such as reachability property and scheduling property.

To exhaustively examine the timing properties of the developed OSEK/VDX applications, model checking [6,7], currently as a reasonable solution, has attracted great attentions in the automobile industry benefiting from its advantages, e.g., model checking is an exhaustive technique, and moreover, many automatic model checkers such as VCC [4], UPPAAL [9], Spin [10] and ESBMC [16] have already been used to verify embedded systems. As to make the existing model checkers successfully verify the timing properties of the OSEK/VDX applications, Libor et al. have proposed a method [21] based on the model checker UPPAAL. In the method, to explicitly simulate the executions of an OSEK/VDX application, all tasks within the application are simulated by concurrent processes (timed automata). In addition, a special concurrent process for OSEK/VDX OS is added in the application model to realize the scheduling behaviours. Although this method is in a position to verify the OSEK/VDX



applications by using UPPAAL, it is often not capable of verifying a large-scale OSEK/VDX application that consists of many tasks, because the constructed application model holds too many concurrent **processes**. In the verification stage, these concurrent **processes** will generate a large number of interleavings which will easily trigger the state space explosion [8].

In this paper, we propose an efficient approach based on the sequentialization idea in order to make UPPAAL more scalable in verifying the timing properties of the large-scale OSEK/VDX applications. The scalability of the existing method is limited because of too many concurrent **processes** included in the application model. To solve this problem, one of impactful ways is to reduce the number of concurrent **processes**. In our approach, we translate a given multi-tasks OSEK/VDX application into a sequential model and then use only one concurrent **process** to simulate the application. Particularly, the OSEK/VDX OS is embedded in the sequential translation algorithm to perform the scheduling behaviours in order to further reduce the states of the application model. Based on these efforts, the application model can be significantly simplified by our approach.

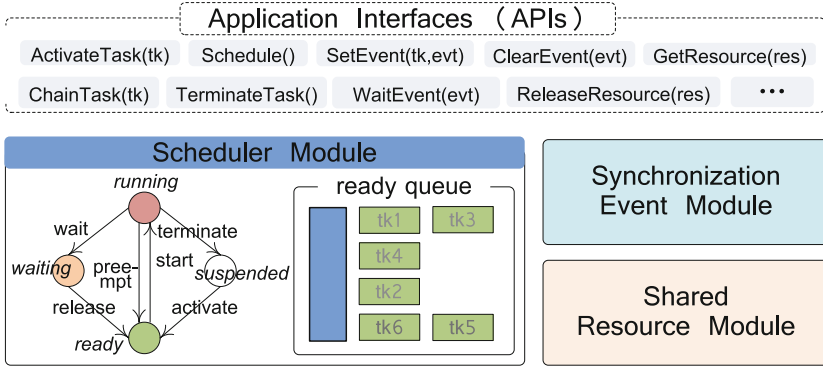
Based on the proposed approach, we have implemented a prototype tool with C++ language and conducted a series of experiments to evaluate the scalability and efficiency of our approach. In the experiments, we firstly use approach to translate the experimental applications into the equivalent sequential models, and then employ model checker UPPAAL to carry out verification. In addition, the existing method proposed by Libor is considered as comparison object. The experimental results indicate that our approach is not only capable of efficiently simplifying OSEK/VDX application models, but also of making the model checker UPPAAL competent in dealing with OSEK/VDX applications with industrial complexity in contrast with the existing method. To the best of our knowledge, our approach is first to apply model checking to examine the timing properties of *deterministic* scheduler based concurrent programs by means of the sequentialization technique. The main contribution of our paper is that the proposed approach can be considered as a guideline to verify the other *deterministic* scheduler based concurrent programs on timing properties, such as round robin based concurrent programs.

The outline of this paper is as follows. The OSEK/VDX OS and a running application are introduced in Sect. 2. The existing checking method is shown in Sect. 3. In Sect. 4, our approach are stated based on the running application. To show the scalability and efficiency of our approach, experiments and evaluation are demonstrated in Sect. 5. We then compare our work with related work in Sect. 6, and put conclusion and future work in the last Section.

## 2 Background of OSEK/VDX

### 2.1 OSEK/VDX OS

The OSEK/VDX standard is a widespread development specification for vehicle-mounted system on a single CPU. It supports the implementation of a vehicle-



**Fig. 2.** Structure of the OSEK/VDX OS and corresponding APIs.

mounted OS and the development of customized multi-tasks applications. In general, as shown in Fig. 2, an OSEK/VDX OS consists of three primary process modules: a scheduler module, a synchronization event module and a shared resource module. In addition, these process modules also provide many useful application interfaces (APIs) to allow developers to operate tasks, synchronization events and shared resources declared in applications. The process modules of the OSEK/VDX OS and corresponding APIs are as follow.

**Scheduler Module.** The OSEK/VDX OS allows developers to define two types of tasks in application: basic task and extended task. A basic task holds three states: *running* state, *suspended* state, and *ready* state. Compared with the basic task, an extended task can take synchronization events, and holds a unique state called *waiting* state. In the scheduling process, a deterministic scheduling policy, which is a static priority scheduling policy with mix-preemptive strategy (Full-preemptive strategy and Non-preemptive strategy), is adopted by the OSEK/VDX OS to conduct the executions of tasks, where a ready queue is used to manage the scheduling order of tasks. Furthermore, the OSEK/VDX scheduler provides several APIs for applications (e.g., *TerminateTask*, *ActivateTask*, *Schedule* and *ChainTask*), and tasks within an application can invoke these APIs to dynamically change the states of tasks. For example, when a task that is currently running invokes API *ActivateTask(tk1)* and the activated task *tk1* is in the *suspended* state, *tk1* will be moved from *suspended* state to *ready* state by the OSEK/VDX scheduler.

**Synchronization Event Module.** The OSEK/VDX OS also supports a synchronization mechanism. The synchronization event module provides several APIs (e.g., *SetEvent*, *WaitEvent* and *ClearEvent*), and tasks within an application can invoke these APIs to implement the synchronous executions.

For example, a running task  $t1$  invokes the API  $WaitEvent(evt1)$ , task  $t1$  is waited until the event  $evt1$  occurs by being set in another task using the API  $SetEvent(t1,evt1)$ .

**Shared Resource Module.** The OSEK/VDX OS adopts the *Priority Ceiling Protocol* to coordinate the task behaviours for accessing shared resources. The shared resource module provides two APIs for applications, i.e.,  $GetResource$  and  $ReleaseResource$ . For example, if a task needs to access a shared resource  $res1$ , it can invoke the APIs  $GetResource(res1)$  and  $ReleaseResource(res1)$  to create a critical section for accessing the shared resource.

## 2.2 Running Application

As shown in Fig. 3, an OSEK/VDX application consists of two files, one is source file, and the other is configuration file. The source file is used to present the concrete behaviors of an application, which can be developed by C programming language. The configuration file is used to configure application, e.g., define tasks, synchronization events and shared resources. In a task configuration, the attribute **Type** is to define the type of the task (Basic or Extended). The attribute **Priority** is to set the priority of the task. **Schedule** is to indicate the scheduling type of the task. If the attribute **Schedule** is set to **Full**, the task can be preempted by higher priority tasks; otherwise, it is impossible. **Autostart** is to specify the initial state of the task. If the attribute is set to **True**, the task starts from *ready* state as its initial state (it will be inserted into the ready queue according to its priority); otherwise, the task starts from *suspended* state.

The execution characteristics of the OSEK/VDX applications are expressly explained by describing the executions of the running application shown in Fig. 3 in this paragraph. There are three tasks *contask*, *plustask* and *minustask* in the running application. When the application starts, since only *contask* is in the *ready* state (the attribute **Autostart** of *contask* is configured as **True**), the OSEK/VDX scheduler moves *contask* from *ready* state to *running* state. When *contask* is running on the processor, the API  $ActivateTask(plustask)$  or  $ActivateTask(minustask)$  will be invoked in **if-else** branches. For instance, if  $ActivateTask(plustask)$  is invoked, the OSEK/VDX scheduler is loaded to respond to this API, and then *plustask* is activated (the OSEK/VDX scheduler moves *plustask* from *suspended* state to *ready* state). At this moment, the currently running task *contask* will be preempted by the activated task *plustask*, because the priority of *contask* is lower than that of *plustask* and the attribute **Schedule** of *contask* is set to **Full**. Then, *plustask* will get processor to run and go to *suspended* state when the API  $TerminateTask()$  is invoked (API  $TerminateTask()$  is used to terminate a task, and the terminated task will be moved from *running* state to *suspended* state by the OSEK/VDX scheduler). When *plustask* is terminated, the OSEK/VDX scheduler will dispatch *contask* to run again from the preempted point. Similarly, if  $ActivateTask(minustask)$  is invoked by *contask*, *minustask* will preempt *contask* to run and terminate itself when the API  $TerminateTask()$  is invoked.

The image shows two side-by-side code editors. The left editor, titled 'Source file', contains C code for a task-based application. It defines a global variable 'fixspeed' and a 'while' loop that reads 'speed' and 'switch' values, and activates 'plustask' or 'minustask' based on the 'switch' value. It also defines three task functions: 'contask', 'plustask', and 'minustask'. The right editor, titled 'Configuration file', defines three task configurations: 'contask' (Basic, Priority=2, Full schedule, Autostart=True), 'plustask' (Basic, Priority=3, Non schedule, Autostart=False), and 'minustask' (Basic, Priority=3, Non schedule, Autostart=False).

```

Source file
int speed, fixspeed;
Task contask() {
    fixspeed=120;
    bool switch=true;
    while(switch==true){
        speed=read_speed();
        if(speed<fixspeed)
            ActivateTask(plustask);
        else
            ActivateTask(minustask);
        switch=read_switch();
    }
    TerminateTask();
}

Task plustask() {
    speed++;
    TerminateTask();
}

Task minustask() {
    speed--;
    TerminateTask();
}

Configuration file
Task contask
{
    Type=Basic;
    Priority=2;
    Schedule=Full;
    Autostart=True;
}

Task plustask
{
    Type=Basic;
    Priority=3;
    Schedule=Non;
    Autostart=False;
}

Task minustask
{
    Type=Basic;
    Priority=3;
    Schedule=Non;
    Autostart=False;
}

```

Fig. 3. Running application.

**Execution Characteristics.** According to the shown executions of the running application, we can find the following execution characteristics.

- Tasks within an OSEK/VDX application are concurrently executed under the scheduling of OSEK/VDX OS, and the running task can be explicitly determined by the OSEK/VDX scheduler.
- Tasks can invoke APIs to change the states of tasks, and the changed task states will dynamically update the scheduling order of tasks.

Due to the concurrency of tasks and dynamic of scheduling, how to exhaustively examine the timing property of the developed OSEK/VDX applications is becoming a challenge in the automotive industry. In the following sections, we will demonstrate two exhaustive methods based on the model checker UPPAAL. The first method feeds UPPAAL with a complex application model. In contrast with the first method, the second method firstly translates a given OSEK/VDX application into a sequential model and then employs UPPAAL to carry out verification.

### 3 The Existing Method for OSEK/VDX Applications

#### 3.1 Non-deterministic and Deterministic Schedulers

In the OSEK/VDX distributed application system, tasks within an application are currently executed and the running task can be explicitly determined by the OSEK/VDX scheduler based on the deterministic scheduling. For example, as

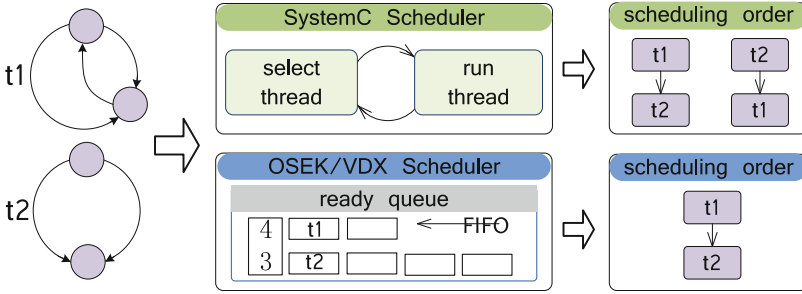


Fig. 4. Example for deterministic and non-deterministic schedulers.

shown in Fig. 4, assuming that an OSEK/VDX application contains two tasks  $t1$  and  $t2$  currently staying in the ready queue, and the priority of  $t1$  is higher than that of  $t2$ , the OSEK/VDX scheduler will make one task scheduling order ( $t1, t2$ ) and  $t1$  is firstly selected as the running task. In general, we call this type of scheduler as a deterministic scheduler.

Compared with the OSEK/VDX scheduler, multi-threaded programs in general, such as SystemC and ANSI-C concurrent programs, usually take a non-deterministic scheduling. Threads in such programs are concurrently executed, but the running thread is hard to be explicitly determined because the scheduler selects the running thread from runnable or ready threads in an arbitrary way. For example, as shown in Fig. 4, suppose there are two threads  $t1$  and  $t2$  in a SystemC multi-threaded program. If these two threads are currently in the runnable or ready state, there exist two possible scheduling orders, one is ( $t1, t2$ ), and the other is ( $t2, t1$ ), since this way of scheduling makes it hard to determine which of  $t1$  and  $t2$  can be the running thread in advance. We usually call this type of scheduler as a non-deterministic scheduler.

In the scope of model checking, many advanced methods [1, 18] have been proposed for verifying the *non-deterministic* scheduler based concurrent programs. However, these methods are not suitable to verify the *deterministic* scheduler based OSEK/VDX applications. If we apply these methods to carry out verification, a large number of unnecessary interleavings of tasks will be checked by model checkers (i.e., the state space verified by the existing methods is larger than the real state space of the target application). Furthermore, due to the unnecessary interleavings, model checkers will often find a spurious bug which makes the verification inexplicit. To overcome this problem, Libor has proposed a method based on the model checker UPPAAL by means of a worthy application model. The details of the method are stated below.

### 3.2 Application Model in UPPAAL

To explicitly verify a given OSEK/VDX application using UPPAAL, the key problem is how to construct an application model to accurately simulate the executions of the application. In the existing method, according to the execution

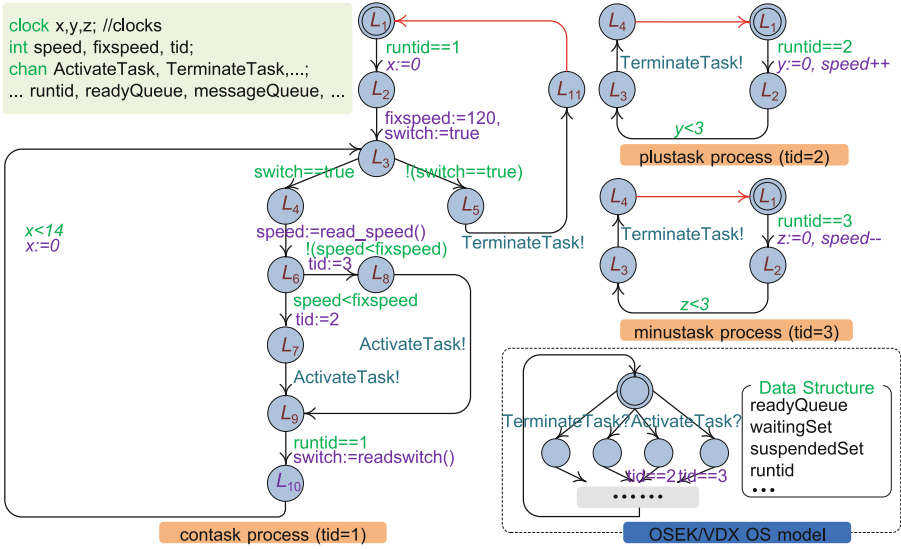


Fig. 5. UPPAAL model for the running application.

characteristics of the OSEK/VDX applications, concurrent processes (timed automaton) are used to simulate each tasks. In particular, a special concurrent process for the OSEK/VDX OS is added in the application model to determine the running task and respond to the APIs invoked from tasks. Based on the application model, all of the possible interleavings of tasks will be verified by UPPAAL, especially the unnecessary interleavings of tasks will be removed by the scheduling of OSEK/VDX OS process in the verification stage.

As shown in Fig. 5, we can use the application model to simulate the running application depicted in Fig. 3 and verify the application by using model checker UPPAAL. The shown application model is a prototype of the real model. We just show the details of the running application, the details of OSEK/VDX OS is omitted, since the real model of the OSEK/VDX is too complex to demonstrated here (the real OSEK/VDX OS model is described in papers [12, 21]). In the shown application model, the guard statements “rntid==1”, “rntid==2”, “rntid==3”, and update statements “tid:=2”, “tid:=3” within the task concurrent processes (timed automata) are assistant statements used to restrict the executions of the tasks that are not the currently running task (the currently running task ID is determined by the OS model). The red transitions are assistant transitions used to return initial locations of tasks for realizing the activated behaviours (i.e., in the OSEK/VDX applications, a terminated task can be activated again by other tasks using the API *ActivateTask(tid)*, the existing method thus adds an assistant transition in task concurrent processes to make tasks return their initial state). In addition, the variables  $x$ ,  $y$  and  $z$  are clocks used to simulate the running time of task instructions. The running time of the

task instructions generally is CPU cycles estimated from CPU specification. The related work for computing the running time can be found in papers [17, 19].

### 3.3 Advantages and Disadvantages of the Existing Method

The advantage of the existing method is that it can explicitly verify the timing properties of the OSEK/VDX applications by using UPPAAL. However, we can sensitively find that the existing method is not scalable to verify a large-scale OSEK/VDX application which contains many tasks, because the method has to use a lot of concurrent **processes** to simulate the executions of the application. In the verification stage, these concurrent **processes** will easily make UPPAAL meet its verification limitation.

## 4 Our Approach for Boosting UPPAAL

### 4.1 Key Idea of Our Approach

To boost UPPAAL more scalable in verifying a large-scale OSEK/VDX application, we propose an efficient approach based on the sequentialization idea. The scalability of the existing method is limited because the constructed application is too complex. In our approach, we tackle this problem by translating an OSEK/VDX application into an equivalent sequential model. Based on the sequential model, we only need one concurrent **process** to simulate the application. In the verification stage, UPPAAL just checks one concurrent **process** rather than multiple **processes** compared with the existing method. The key process of the sequential translation is that we symbolically execute an application by means of an extended directed graph. In order to keep the equivalence between the original application and the sequentialized model, the OSEK/VDX OS model is embedded in the sequential algorithm to dispatch tasks and respond to the APIs invoked from tasks.

### 4.2 Sequential Translation

There are two problems that should be addressed when we translate a multi-tasks application into an equivalent sequential model, one is how to explicitly perform the scheduling behaviours of OSEK/VDX OS, and the other is how to compute the sequential model. In our approach, in order to explicitly perform the scheduling behaviours of the OSEK/VDX OS, we embed an OSEK/VDX OS model in the sequential algorithm for dispatching tasks and responding to the APIs invoked from tasks. The embedded OS model is an extended version shown in paper [21], consisting of two components. The first one is a set of data structures used to record the scheduling data such as the states of tasks. The second one is a set of functions used to dispatch tasks and respond to the APIs based on the data structures. In addition, as to equivalently compute the sequential model of an OSEK/VDX application, an extended directed graph shown in

---

**Input:** Application: Timed automata of tasks, configuration file  
**Output:** Extended directed graph  $G$   
create a start node  $v_0$  and initialize a set  $V$ ;  
initialize  $p$  and  $D$  in node  $v_0$  with the initial locations of task timed automata and configuration file,  $V := \{v_0\}$ ;  
call OS model to determine the currently running task  $t$  in node  $v_0$  based on  $D$  of node  $v_0$ ;  
create a node  $v_1$  and a set  $V'$ ,  $v_1 := v_0$ ,  $V' := \{v_1\}$ ;  
start the sequential translation from node  $v_1$  in directed edge  $(v_0, v_1)$ ;  
**while**  $V' \neq \emptyset$  **do**  
     $v \in V'$ ,  $V' := V' \setminus \{v\}$ ;  
    **if** running task  $t$  in node  $v$  is not null **then**  
        explore transitions of task  $t$  from the current location in symbolic way;  
        **if** explored transition holds an API **then**  
            create a new node  $v'$ , where  $v' := v$ ;  
            map the explored transition in directed edge  $(v, v')$ ;  
            update  $p$  of node  $v'$  with the current location of running task  $t$  in node  $v$ ;  
            **call OS model to respond to the API and determine running task  $t$  in node  $v'$  based on  $D$  of node  $v'$** ;  
             $V' := V' \cup \{v'\}$ ;  
        **end**  
        **if** explored transitions are branch statements **then**  
            according to the number  $m$  of branches, create  $m$  new nodes  $v'_1, \dots, v'_m$ , where  
             $v'_1 := v, \dots, v'_m := v$ ;  
            map the explored branch statements in directed edges  $(v, v'_1), \dots, (v, v'_m)$   
            respectively;  
            update  $p$  of node  $v'_1, \dots, v'_m$  with the current location of running task  $t$  in node  
             $v$ ;  
             $V' := V' \cup \{v'_1, \dots, v'_m\}$ ;  
        **else**  
            create a new node  $v'$ , where  $v' := v$ ;  
            map the explored transition in directed edge  $(v, v')$ ;  
            update  $p$  of node  $v'$  with the current location of running task  $t$  in node  $v$ ;  
             $V' := V' \cup \{v'\}$ ;  
        **end**  
        **forall** the  $v_i \in V'$  **do**  
            **if**  $v_i = v_j \in V$ , then change the relationship of directed edge  $(v, v_i)$  to  $(v, v_j)$ ,  
             $V' := V' \setminus \{v_i\}$ ; **Otherwise**,  $V := V \cup \{v_i\}$  (where,  $v_i = v_j$  means that  $p$  and  $D$  in  
            node  $v_i$  are equal to  $p$  and  $D$  in node  $v_j$  respectively);  
        **end**  
    **end**  
**end**  
**return**  $G$ ;

---

**Algorithm 1.** Key processes of the sequential translation

Definition 1 is employed to carry out sequential translation. In the sequential translation, we use the extended directed graph to execute the application in symbolic way and call embedded OS model to determine the running task when meeting an API, i.e., the context switch of tasks in OSEK/VDX applications occurs at the invoked points of APIs.

**Definition 1.** The extended directed graph  $G$  is a triple  $G = (V, v_0, E)$ . Where,  $V$  is a set of nodes with the start node  $v_0 \in V$ . A node  $v \in V$  consists of two variables  $p$  and  $D$ ,  $p$  is an array with the size equaling to the task number used to record the current locations of timed automata of tasks, and  $D$  is a set of data structures used to store the scheduling data such as the states of tasks.  $E \subseteq V \times V$



is a set of directed edges used to map the transitions of timed automata of tasks, such as an action, a guard or a set of clocks to be reset.

Based on the extended directed graph and the embedded OSEK/VDX OS model, the key processes of the sequential translation are formalized in Algorithm 1. The algorithm accepts the task timed automata used in UPPAAL [9] and configuration file as inputs. In the sequential translation processes, it does not compute the values of variables, instead, it just explores transitions of task timed automata according to the executions of tasks by using the defined extended directed graph. In the extended directed graph, nodes are in charge of recording the current locations of task timed automata and the scheduling data. When the algorithm starts the explorations from a node, if the explored transitions in task automata are sequential statements and branch statements, it will create several new nodes in the extended directed graph and then maps the explored transitions in directed edges. If the explored transition holds an API, the algorithm firstly passes the scheduling data recorded in the start node to the embedded OS model and then calls embedded OS model to respond to the API and determine the next running task. Once the OS model completes its execution, a new node is created and the current scheduling data is recorded by the new node, where we stipulate that the invoked APIs monopolize one transition in task timed automata. Particularly, if an OSEK/VDX application holds loops, it will usually go back to a previous state. In the sequential translation processes, the algorithm will construct a cyclic in the extended directed graph  $G$  in order to reduce the same states of the application to be created. In the algorithm, if the task locations  $p$  and the scheduling data  $D$  in a new node  $v_i$  are equal to that of an old node  $v_j$ , a cyclic will be constructed. Based on the sequentialization translation, we can obtain a sequential model that is equivalent to the original application. The reason is that all execution traces in the original application are held by the sequential model, especially these execution traces are under the scheduling.

### 4.3 Example

To facilitate the understanding of the sequential translation of Algorithm 1, we demonstrate an example to show the details by sequentializing the running application. In the shown example, the assistant statements and transitions used in the existing method are omitted, because our algorithm does not need them in the sequential translation. In addition, since the responding behaviours for APIs invoked from tasks have been realized by the embedded OSEK/VDX OS model in the sequential transition, the APIs in the task automata are annotated in the sequential model.

As shown in Fig. 6, in the first step, the algorithm creates a start node  $v_0$ , and then initializes  $v_0$  with the initial locations of task timed automata and configuration file (in node  $v_0$ , all tasks start from initial locations  $L_1$  shown in Fig. 5, and *contask* is in the *ready* state, *plustask* and *minustask* are in the

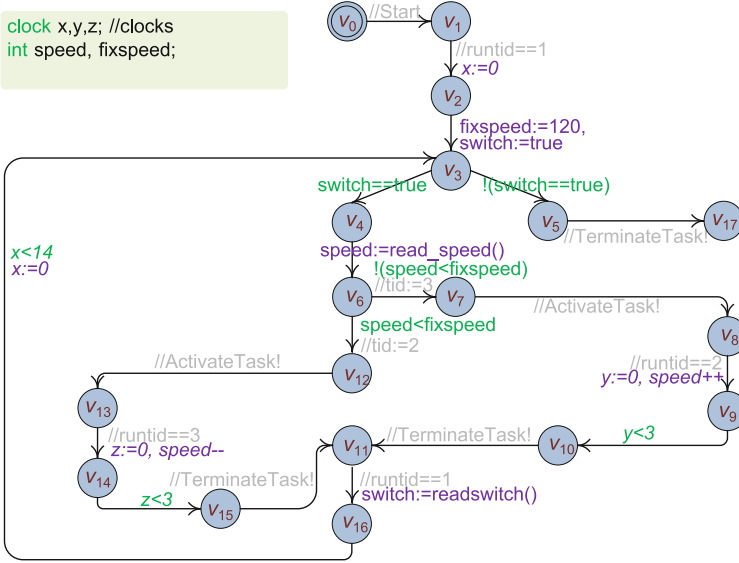


Fig. 6. Sequential model for the running application.

*suspended* state). In the second step, the algorithm calls OS model to determine the currently running task based on the data  $D$  in node  $v_0$  and then starts the sequential translation. In node  $v_1$ , *contask* becomes the running task. Thus, the algorithm successively explores the transitions of *contask* and maps the explored statements in directed edges of  $G$ , e.g., two nodes  $v_2$  and  $v_3$  are created, and the explored statements are mapped in directed edges  $(v_1, v_2)$  and  $(v_2, v_3)$ . When the algorithm explores task transitions from node  $v_3$  (*contask* locates at  $L_3$ , *plustask* and *minusstask* locate at  $L_1$ ), it will meet the condition statements of **while** loop. Consequently, two node  $v_4$  and  $v_5$  are created, and the loop condition statement and the negative condition statement are mapped in directed edges  $(v_3, v_4)$  and  $(v_3, v_5)$ , respectively. To explore task transitions from node  $v_5$ , the API *TerminateTask* is invoked by *contask*. The embedded OS model is called to respond to the API and compute the states of tasks. Then, the API is mapped in the edge  $(v_5, v_{17})$ . At this moment, all tasks within the application are in the *suspended* state in node  $v_{17}$ . If the algorithm explores task transitions from node  $v_4$ , according to the executions of *contask*, five nodes  $(v_6, v_7, v_8, v_{12}, v_{13})$  are created for mapping the corresponding task transitions, e.g., the APIs *ActivateTask(plustask)* and *ActivateTask(minustask)* are mapped in directed edges  $(v_7, v_8)$  and  $(v_{12}, v_{13})$ , respectively. In edge  $(v_7, v_8)$ , the API *ActivateTask(plustask)* is invoked. The algorithm calls the embedded OS model to respond to the API and compute the states of tasks by using the scheduling data in node  $v_8$ . *contask* is presently preempted by *plustask* in node  $v_8$ , and then the algorithm creates three nodes  $v_9, v_{10}$  and  $v_{11}$  to successively map the transitions of *plustask*. As seen in Fig. 6, the mapped statement in directed

edge  $(v_{10}, v_{11})$  is the API *TerminateTask*, OS model is then called for responding to the API and computing the states of tasks in node  $v_{11}$  (in node  $v_{11}$ , *contask* becomes the running task again). Accordingly, the algorithm also maps the task statements of *minustask* in directed edges  $(v_{13}, v_{14})$ ,  $(v_{14}, v_{15})$  and  $(v_{15}, v_{11})$  from node  $v_{13}$ . After that, the algorithm creates one node to continue the executions of *contask* from node  $v_{11}$ , and the explored transitions are mapped in the corresponding directed edges. In the light of the sequential transition performed by the algorithm, the OSEK/VDX applications can be equivalently translated into the corresponding sequential models.

#### 4.4 Advantages and Disadvantages of Our Approach

In the existing method, the application model of an OSEK/VDX application is composed of task concurrent **processes** and OSEK/VDX **process**. In contrast with the existing method, our approach translates an OSEK/VDX application into a sequential model. Thus, we can use only one concurrent **process** to simulate the application. In addition, in our approach, the OSEK/VDX OS is embedded in the sequential algorithm to dispatch tasks and respond to the invoked APIs, which can significantly reduce the state space of the application model. These efforts make our approach more scalable in dealing with the OSEK/VDX applications. However, since the OSEK/VDX OS model is not included in the application model in our approach, the scheduling time for responding different APIs cannot be directly estimated if we intend to verify the timing property about schedule. To overcome this shortcoming, we can use the worst-case responding time of the OSEK/VDX OS to simulate the scheduling time. Even though the scheduling time is an approximate time for different APIs, the target system will straitly satisfy its design or specification if it passes the verification, because the real time is less than the worst-case time.

## 5 Experiments and Discussion

### 5.1 Experiments

We implemented a prototype tool according to the proposed approach with C++ programming language. Based on the implemented tool, we have conducted a series of experiments to evaluate the efficiency and scalability of our approach. The task number and API number are the primary influence factors of the performance of our approach. Thus, our approach is comprehensively investigated on the OSEK/VDX applications by selecting experimental systems with the different task and processor numbers as benchmarks. The selected OSEK/VDX applications are realistically represented by additionally taking into account the non-preemptive scheduling behaviours, full-preemptive scheduling behaviours, mix-preemptive scheduling behaviours, synchronous behaviours, and accessing shared resource behaviours. In the selected benchmarks, each task within applications takes a clock and holds at least 100 states. In addition, the existing

**Table 1.** Comparison: existing checking method *VS* Our approach

			Existing method (UPPAAL)			Our approach					
						Sequential translation		UPPAAL			
Benchmark	#t	#API	#s	Time	Mb	Time	Mb	#s	Time	Mb	
Non-preemption	6	11	5651	20.1	27.7	0.18	3.19	1321	10.4	13.5	
	8	14	27253	114.3	113.9	0.26	3.64	5431	20.4	86.4	
	10	17	143835	521.7	423.2	0.30	4.11	23812	98.6	214.5	
	18	35	-	T.O	-	0.60	5.72	73853	243.8	517.2	
Full-preemption	4	61	95671	78.1	61.7	0.14	2.98	19176	38.4	61.9	
	6	101	134371	509.8	398.4	0.19	3.34	21328	97.1	104.7	
	9	161	246990	887.1	984.7	0.29	4.05	44523	114.7	241.7	
	13	241	-	-	M.O	0.45	5.14	87246	345.2	507.1	
Mix-preemption	5	101	213541	804.9	945.2	0.18	3.21	24518	99.4	108.7	
	9	161	-	-	M.O	0.30	4.22	42597	189.1	204.2	
	13	241	-	-	M.O	0.45	5.23	63868	357.9	412.3	
	13	313	-	-	M.O	0.41	4.78	88390	687.2	779.6	
Synchronization	5	14	5443	21.3	28.1	0.18	3.16	1421	11.6	14.2	
	8	23	24159	109.8	107.5	0.26	4.01	5214	19.2	84.1	
	11	32	210241	607.2	514.9	0.42	4.53	14157	45.1	62.7	
	12	42	392329	879.8	798.3	0.45	5.11	27891	114.6	124.3	
Shared resource	2	4	13907	54.9	48.6	0.21	3.41	2479	13.4	19.8	
	9	320	-	-	M.O	0.34	4.63	37589	118.2	267.9	
	12	480	-	-	M.O	0.41	5.84	88416	647.2	745.1	
	20	480	-	-	M.O	0.44	4.98	272614	998.2	994.1	

checking method is considered as comparison object to show whether our approach can make UPPAAL more scalable in verifying OSEK/VDX applications.

All experiments are conducted on the Intel Core(TM)i7-3770 CPU with 8G RAM, and we set the time limit and memory limit to 1000 s and 1 GB, respectively. In addition, in order to investigate the scalability of our approach, we thoroughly evaluated our approach by not specifying any property in the experiments to allow UPPAAL to explore the states of the target benchmarks as much as possible. The experiment results are listed in Table 1. In the table, #t is the number of tasks, #API is the times of invoked APIs by tasks, #s is the number of explored states by UPPAAL. “Mb” and “Time” are the memory consumption and time consumption measured in Mbyte and seconds, respectively. M.O. and T.O. stand for that UPPAAL runs out of time and memory, respectively.

## 5.2 Discussion

The experimental results shown in Table 1 indicate that the existing checking method fails to verify the benchmarks which contain a number of tasks and APIs (e.g., lines 4, 8, 10 and 18). This is because, in the existing checking method, the application model holds too many concurrent processes, thereby easily causing the state space to experience exponential growth in the verification stage. In addition, in the method, when an API is invoked by the running task, the states of OSEK/VDX OS are explored, and this significantly increases the state space

if an application holds a number of APIs. These drawbacks seriously limit the efficiency and scalability of this method.

In contrast with the existing checking method, our approach can successfully verify these benchmarks with lower costs (time and memory) and lesser states. This is because, based on the sequential translation of our approach, the model checker UPPAAL only verifies one concurrent **process** rather than multiple concurrent **processes**. In addition, we embedded the OSEK/VDX OS model in the sequential translation algorithm to explicitly dispatch tasks and respond to the invoked APIs, thereby resulting in the sequential model only holding the states of the given application. These efforts can significantly improve the efficiency and scalability of UPPAAL in verifying large-scale OSEK/VDX applications.

Based on the experimental results, we can optimistically find that our approach can be used to verify an OSEK/VDX application with industrial complexity, because our approach can successfully verify twenty tasks. To the best of our experiences, a realistic OSEK/VDX application usually contains less-than twenty tasks. In addition, based on the conducted experiments, we find that the number of clocks taken by tasks is also an influence factor for limiting the performance of our approach. In the future, the technique for reducing clocks [3] will be used in order to make our approach more efficient and scalable.

## 6 Related Work

Currently, there have been many methods that apply model checking techniques to verify the OSEK/VDX standard based software systems. In the scope of checking developed OSEK/VDX OS, Chen and Aoki proposed a method [14] to generate the highly reliable test-cases for checking whether the developed OS conforms to the OSEK/VDX OS standard based on the model checker Spin. In addition, Choi presented a method [22] to convert an open OSEK/VDX OS named Trampoline [13] into formal models, and an incremental verification approach is proposed to carry out the verification. Furthermore, a CSP-based approach for checking the code-level OSEK/VDX OS is addressed in the paper [23]. All of these related works are different from our work, because our approach focuses on the developed OSEK/VDX applications.

For the developed OSEK/VDX applications, Zhang shows a method [12] based on the model checker Spin. The method is similar as Libor's one, but focuses on the safety properties. The scalability of the method is also limited because of too many details of the OSEK/VDX OS and concurrent **processes** included in the application model. To cut the states of OSEK/VDX OS from application model, a new technique named EPG [11] is proposed based on the SMT-based bounded model checking [2]. Even so, the method is not efficient to check the OSEK/VDX applications which holds a lot of branches, since the method will spend much time exploring all of the execution paths in order to construct the corresponding transition system (application model), which will seriously slow down the performance of the method. Compared with EPG technique, our approach uses an extended directed graph to execute OSEK/VDX

applications in symbolic way rather than exploring execution paths, which is more efficient than the EPG technique.

To verify concurrent programs with a scheduler, Liu and Joseph proposed a transformational method [24] to specification and verification of concurrent application programs executing on systems with limited resources. The method allows the real-time and fault-tolerance requirements to be specified and verified in the traditional theory refinement and temporal verification in a single notation. However, the method does not consider the issue of a possible implementation of the transformation for any existing operating system, and to the best of our knowledge there does not exist any tool support to the transformational approach. In contrast with the method, our approach focuses on a realistic system and the sequential translation is different from this method.

In the field of verifying concurrent programs using sequentialization technique, several methods have been proposed such as [5, 20]. However, the existing methods focus on the non-deterministic scheduler based concurrent programs such as SystemC and ANSI-C, which are not suitable to sequentialize the deterministic scheduler based OSEK/VDX application. In the existing methods, random numbers are appended in the target programs to simulate the non-deterministic scheduling behaviours, and several assistant functions included in the sequential model are used to perform the API behaviours. Compared with the existing methods, although our approach is based on the sequentialization idea, we focus on the deterministic scheduler based concurrent programs and uses an extended directed graph to compute the sequential models of the OSEK/VDX applications. Furthermore, the API behaviours are not included in sequential model, which are performed by the embedded OS model. These techniques are quite different from the existing methods.

## 7 Conclusion and Future Work

The timing properties of the OSEK/VDX applications is really hard to be exhaustively examined. In this paper, we proposed an efficient approach to overcome this problem by using model checker UPPAAL. In the proposed approach, a sequential translation technique is used to simplify the application model for making UPPAAL more scalable in verifying the complex OSEK/VDX applications. We have evaluated the proposed approach based on a series of experiments. The experimental results indicate that our approach is not only capable of efficiently simplifying the application models, but also of making UPPAAL competent in dealing with the OSEK/VDX applications with industrial complexity. In the future, we attempt to apply the clock reduction technique in our approach to make the approach more scalable in handling more complex OSEK/VDX applications. In addition, we intend to extend our approach in verifying other deterministic scheduler based concurrent programs.

**Acknowledgement.** This work is supported by the National Science Foundation of China (Grants No.61602224) and the Fundamental Research Funds for the Central Universities (Grants No.lzujbky-2016-142 and No.lzujbky-2016-k07) and Tianjin Key Laboratory of Advanced Networking (TANK), School of Computer Science and Technology, Tianjin University, Tianjin China, 300350.

## References

1. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: a software model checking approach. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 51–59 (2010)
2. Armin, B., Clarke, E.M., Zhu, Y.: Bounded model checking. *Adv. Comput.* **58**(11), 117–148 (2003)
3. Daws, C., Yovine, S.: Reducing the number of clock variables of timed automata. In: 17th IEEE Real-Time Systems Symposium, p. 73 (1996)
4. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03359-9\\_2](https://doi.org/10.1007/978-3-642-03359-9_2)
5. Campana, D., Cimatti, A., Narasamdya, I., Roveri, M.: An analytic evaluation of SystemC encodings in promela. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 90–107. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22306-8\\_7](https://doi.org/10.1007/978-3-642-22306-8_7)
6. Clarke, E.M., Emerson, E.A., Sifakis, J.: Model checking: algorithmic verification and debugging. *Commun. ACM* **152**(11), 74–84 (2009)
7. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **16**(5), 1512–1542 (1994)
8. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Meyer, B., Nordio, M. (eds.) LASER 2011. LNCS, vol. 7682, pp. 1–30. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-35746-6\\_1](https://doi.org/10.1007/978-3-642-35746-6_1)
9. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30080-9\\_7](https://doi.org/10.1007/978-3-540-30080-9_7)
10. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Lucent Technologies Inc., Bell Laboratories, Boston (2003)
11. Zhang, H., Aoki, T., Lin, X., et al.: SMT-based bounded model checking for OSEK/VDX applications. In: 20th Asia-Pacific Software Engineering Conference, pp. 307–314 (2013)
12. Zhang, H., Aoki, T., Chiba, Y.: A spin-based approach for checking OSEK/VDX applications. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2014. CCIS, vol. 476, pp. 239–255. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-17581-2\\_16](https://doi.org/10.1007/978-3-319-17581-2_16)
13. Bechenec, J., Briday, M., Faucou, S., et al.: Trampoline-an open source implementation of the OSEK/VDX RTOS specification. In: 11th International Conference on Emerging Technologies and Factory Automation, September 2006
14. Chen, J., Aoki, T.: Conformance testing for OSEK/VDX operating system using model checking. In: Proceedings of the 18th Asia-Pacific Software Engineering Conference, pp. 274–281 (2011)

15. Lemieux, J.: Programming in the OSEK/VDX Environment. CMP, Lawrence (2001)
16. Morse, J., Ramalho, M., Cordeiro, L., Nicole, D., Fischer, B.: ESBMC 1.22. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 405–407. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54862-8\\_31](https://doi.org/10.1007/978-3-642-54862-8_31)
17. Sun, J., Liu, Y., Song, S., Dong, J.S., Li, X.: PRTS: an approach for model checking probabilistic real-time hierarchical systems. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 147–162. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-24559-6\\_12](https://doi.org/10.1007/978-3-642-24559-6_12)
18. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: The International Conference on Software Engineering (ICSE), pp. 331–340, May 2011
19. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22110-1\\_47](https://doi.org/10.1007/978-3-642-22110-1_47)
20. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: a lazy sequentialization tool for C. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 398–401. Springer, Heidelberg (2014). [https://doi.org/10.1007/978-3-642-54862-8\\_29](https://doi.org/10.1007/978-3-642-54862-8_29)
21. Libor, W., Jan, K., Zdenek, H.: Case study on distributed and fault tolerant system modeling based on timed automata. *J. Syst. Softw.* **82**(10), 1678–1694 (2009)
22. Choi, Y.: Safety analysis of trampoline os using model checking: an experience report. In: Proceedings of the IEEE 22nd International Symposium on Software Reliability Engineering, pp. 200–209, November 2011
23. Huang, Y., Zhao, Y., Zhu, L., et al.: Modeling and verifying the code-level OSEK/VDX operating system with CSP. In: Fifth International Symposium on Theoretical Aspects of Software Engineering, pp. 142–149 (2011)
24. Liu, Z., Joseph, M.: Specification and verification of fault-tolerance, timing, and scheduling. *ACM Trans. Program. Lang. Syst.* **21**(1), 46–89 (1999)





# The Complexity of Linear-Time Temporal Logic Model Repair

Xiuting Tao and Guoqiang Li<sup>(✉)</sup>

School of Software, Shanghai Jiao Tong University, Shanghai, China  
{xiutingtao, li.g}@sjtu.edu.cn

**Abstract.** We propose the model repair problem of the linear-time temporal logic. Informally, the repair problem asks for a minimum set of states in a given Kripke structure  $M$ , whose modification can make the given LTL formula satisfiable. We will exemplify the application of the model and study the computational complexity of the problem. We will show the problem can be solved in exponential time but remains NP-hard even if  $k$  is a constant.

## 1 Introduction

Model checking is the problem of deciding the satisfiability of a logical formula expressing some desired specification for a given system model. It is an automatic technique for verifying finite state concurrent systems [8]. It has been developed in the context of temporal logic formula for the Kripke structure that represents finite-state system, named “*temporal logic model checking*” [19]. The temporal logic model checking finds its broad applications in verification, and has been a very active field of research in the last three decades.

Important temporal logics in the verification include linear-time temporal logic (LTL) [12], computational tree logic (CTL) [6], CTL\* (a superset of CTL and LTL) [10], along with their fragments and some other extensions. The computational complexity of these model checking has been thoroughly studied in the literature.

In order to understand the system behaviors and obtain more feedback or application-specific information, a lot of work has extended the standard algorithmic technique for variants of model checkings. In [3], Bruns and Godefroid addressed a 3-valued interpretation to modal logic formulas on model structures, in which beyond the values of *true* and *false*, the third value  $\perp$  (means *unknown*) is defined to represent incomplete state spaces and logic formulas. They also presented a 3-valued CTL model checking algorithm. Chan introduced temporal-logic queries in which some temporal-logic formula could be modified for a special symbol, named a placeholder appearing exactly once and defined a class of CTL queries in [5]. In [7], Clarke et al. devised new symbolic techniques which analyze counterexamples and refine the abstract model correspondingly. They also presented a refinement algorithm.

We shall focus on one of the most fundamental fragments of logic, the propositional linear-time temporal logic, proposed in [12]. It is historically the first temporal logic that has been used in formal specification and verification of non-terminating computer programs [1, 14]. Moreover, LTL is the most commonly used specification logic for reactive systems [17]. Like propositional model counting generalizes SAT, LTL model counting introduces “quantitative” extensions of model checking and synthesis [2, 18, 20]. In the field of verification, model repair could be used to determine not only the existence of computations that violate the specification, but also the number of such modified states. For example, in a communication system, where messages are lost (with some probability) in the channel, it is typically not necessary (or even possible) to guarantee a 100% correct transmission. Instead, the number of executions that lead to a message loss is a good indication of the quality of the implementation [11].

We propose the *model repair* with states and transition relations problem for safety specifications expressed in LTL [15]. The LTL model repair problem is described as computing the minimum set of states in a given Kripke structure  $M$  that its atomic propositions can be modified to satisfy a given LTL formula. We shall use this problem to formulate applications including verifying concurrent programs. We also extend classic model checking algorithms for LTL to this problem and demonstrate that the problem can be solved in singly-exponential time. In addition, we prove that even for constant number of modified states, the problem is NP-hard.

The rest of the paper is organized as follows. In Sect. 2, we briefly introduce some basic notions. In Sect. 3, we explain an extended definition and algorithm of LTL model checking, as well as model repair of LTL that modifies states in the model, while in next Sect. 4, we analyze the hardness for the decision version of LTL model repair problem. In Sect. 5, we show some results of another version of LTL model repair that modifies transition relations in the model. In Sect. 6, we give some examples to support the applicability of our approach. We conclude the paper in Sect. 7.

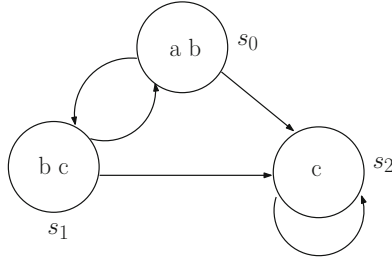
## 2 Preliminaries

Some basic definitions and notes [8] are introduced in this section.

### 2.1 Kripke Structure

**Definition 1.** *Let  $AP$  be a set of atomic propositions. A Kripke structure  $M$  over  $AP$  is a quadruple  $M = (S, S_0, R, L)$  where*

1.  $S$  is a finite set of states.
2.  $S_0 \subseteq S$  is the set of initial states.
3.  $R \subseteq S \times S$  is a transition relation that must be total, that is, for every state  $s \in S$  there is a state  $s' \in S$  such that  $R(s, s')$
4.  $L : S \rightarrow 2^{AP}$  is a function that labels each state with the set of atomic propositions true in that state.



**Fig. 1.** An example of Kripke structure.

*Example 1.* An example of Kripke structure  $M = (S, S_0, R, L)$  is given in Fig. 1, where  $AP = \{a, b, c\}$ ,  $S = \{s_0, s_1, s_2\}$ ,  $S_0 = \{s_0\}$ ,  $R = \{(s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_2), (s_2, s_2)\}$ ,  $L(s_0) = \{a, b\}$ ,  $L(s_1) = \{b, c\}$ ,  $L(s_2) = \{c\}$ .

We are not always concerned with the set of initial states  $S_0$ . In such cases, we will omit this set of states from the definition. A *path* in the structure  $M$  from a state  $s$  is an infinite sequence of states  $\pi = s_0 s_1 s_2 \dots$  such that  $s_0 = s$  and  $R(s_i, s_{i+1})$  holds for all  $i \geq 0$ .

## 2.2 Linear-Time Temporal Logic

Linear-time temporal logic (LTL) is proposed by Pnueli in 1977 [17]. The language of LTL contains an infinite set of propositional variables  $Var = \{p_1, p_2, \dots\}$ , the Boolean connectives  $\neg$ ,  $\vee$  and  $\wedge$ , and the temporal operators.

The LTL formulas are defined by the following expression:

$$\varphi := p | \neg\varphi | (\varphi_1 \vee \varphi_2) | (\varphi_1 \wedge \varphi_2) | X\varphi | F\varphi | G\varphi | \varphi_1 U \varphi_2 | \varphi_1 R \varphi_2$$

We also define  $\top := (\neg p \vee p)$ ,  $\perp := (\neg p \wedge p)$ . We adopt the usual convention about omitting parentheses. For every formula  $\varphi$  and every  $n \geq 0$ , we inductively define the formula  $X^n\varphi$  as follow:  $X^0\varphi := \varphi$ ,  $X^{n+1}\varphi := XX^n\varphi$ .

For temporal logic, there are five basic operators:

- $X$  (“Next”): requires that a property holds in the second state of the path.
- $F$  (“Finally”): is used to assert that a property will hold at some state of the path.
- $G$  (“Globally”): specifies that a property holds at every state on the path.
- $U$  (“Until”): is used to combine two properties. It holds if there is a state on the path where the second property holds and at every preceding state on the path, the first property holds.
- $R$  (“Release”): is the logical dual of the  $U$ . It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

LTL formulas are evaluated with respect to paths. The syntax of state formulas and path formulas are given by the following rules:

- If  $p \in AP$ , then  $p$  is a state formula.
- If  $f$  and  $g$  are state formulas, then  $\neg f$ ,  $f \vee g$  and  $f \wedge g$  are state formulas.
- If  $f$  is a path formula, then  $E f$  and  $A f$  are state formulas.
- If  $f$  is a state formula, then  $f$  is also a path formula.
- If  $f$  and  $g$  are path formulas, then  $\neg f$ ,  $f \vee g$ ,  $f \wedge g$ ,  $X f$ ,  $F f$ ,  $G f$ ,  $f U g$  and  $f R g$  are path formulas.

### 2.3 LTL Model Checking

The behavior of a finite-state parallel program can be modeled as a Kripke structure. Each infinite path starting from the initial states represents a possible execution of the individual processes in the Kripke structure. In most cases, the correctness requirements of the system can be expressed by a propositional linear-time temporal logic formula. The system would be correct if only if every possible execution sequence satisfies this formula; that is, every sequence starting at the initial states in the corresponding Kripke structure satisfies the formula. For these reasons, LTL model checking is important in verifying finite-state parallel programs.

**Definition 2 (LTL model checking).** *Given a Kripke structure  $M = (S, S_0, R, L)$ , a state  $s \in S_0$  and an LTL formula  $g$ , whether there is an infinite path  $\pi$  in the  $M$  starting from  $s$  such that  $M, s \models E g$ .*

$M = (S, S_0, R, L)$  is a Kripke structure with  $s \in S_0$  and let  $g$  be a linear-time temporal logic formula. Thus,  $g$  is a restricted path formula in which the only state subformulas are atomic propositions. We hope to determine if  $M, s \models A g$ . Notice that  $M, s \models A g$  if and only if  $M, s \models \neg E \neg g$ . Consequently, it is sufficient to be able to check the truth of formulas of the form  $E f$  where  $f$  is a restricted path formula.

In the LTL model checking problem,  $|S|$  denotes the size of the states in the Kripke structure, as the  $|R|$  for the size of transition relations, and  $|f|$  for the length of LTL formula  $f$ . In general, this problem is PSPACE-complete. The model-checking problem is NP-hard for formulas of the form  $E f$  where  $f$  is a restricted path formula.

**Closures and Atoms.** Let  $M = (S, S_0, R, L)$  be a Kripke structure and let  $f$  be a restricted path formula. Let  $f$  be a restricted path formula. It is sufficient to consider only the temporal operators  $X$  and  $U$ , since  $F f = \text{True } U f$ ,  $G f = \neg F \neg f$  and  $f_1 R f_2 = \neg[\neg f_1 U \neg f_2]$ .

The closure of  $f$ ,  $CL(f)$  is the smallest set of formulas containing  $f$  and satisfying:

- $\neg f_1 \in CL(f) \iff f_1 \in CL(f)$
- $f_1 \vee f_2 \in CL(f) \Rightarrow f_1, f_2 \in CL(f)$
- $X f_1 \in CL(f) \Rightarrow f_1 \in CL(f)$
- $\neg X f_1 \in CL(f) \Rightarrow X \neg f_1 \in CL(f)$
- $f_1 U f_2 \in CL(f) \Rightarrow f_1, f_2, X[f_1 U f_2] \in CL(f)$

(In the above, we identify  $\neg\neg f_1$  with  $\neg f_1$ ). Notice that the size of  $CL(f)$  is linear in the size of  $f$ .

A normal atom is a pair  $A = (s_A, K_A)$  with  $s_A \in S$  and  $K_A \subseteq CL(f) \cup AP$  such that:

- For each proposition  $p \in AP$ ,  $p \in K_A \Leftrightarrow p \in L(s_A)$
- For every  $f_1 \in CL(f)$ ,  $f_1 \in K_A \Leftrightarrow \neg f_1 \notin K_A$
- For every  $f_1 \vee f_2 \in CL(f)$ ,  $f_1 \vee f_2 \in K_A \Leftrightarrow f_1 \in K_A \text{ or } f_2 \in K_A$
- For every  $\neg X f_1 \in CL(f)$ ,  $\neg X f_1 \in K_A \Leftrightarrow X\neg f_1 \in K_A$
- For every  $f_1 U f_2 \in CL(f)$ ,  $f_1 U f_2 \in K_A \Leftrightarrow f_2 \in K_A \text{ or } f_1, X[f_1 U f_2] \in K_A$ .

The set of all atoms is denoted by  $At$ . A graph  $G(At, E)$  is constructed with the set of atoms as the set of vertices and the edges defined as:

$$(A, B) \in E \Leftrightarrow \begin{cases} (s_A, s_B) \in R; & (1) \\ \forall (X f_1) \in CL(f), X f_1 \in K_A \Leftrightarrow f_1 \in K_B & (2) \end{cases}$$

**Model Checking Algorithms.** In the constructed graph, some related definitions and lemmas introduced in the following are configured to display the model checking algorithms by tableau.

**Definition 3.** *An eventuality sequence is an infinite path  $\pi$  in  $G$  such that  $f_1 U f_2 \in K_A$  for some atom  $A$  on  $\pi$ , then there exists an atom  $B$ , reachable from  $A$  along  $\pi$ , such that  $f_2 \in K_B$ .*

**Lemma 1.**  *$M, s \models E f$  iff there exists an eventuality sequence starting at an atom  $(s, K)$  such that  $f \in K$ .*

( $\Leftarrow$ ) the “only if” case is very trivial.

( $\Rightarrow$ ) assume that there is an eventuality sequence  $(s_0, K_o), (s_1, K_1), \dots$  starting at  $(s, K) = (s_0, K_o)$  with  $f \in K$ . By definition,  $\pi = s_0, s_1, \dots$  is a path in  $M$  starting at  $s = s_0$ . We want to show that  $\pi \models f$ .

Actually, we prove a stronger claim: for every  $g \in CL(f)$  and every  $i \geq 0$ ,  $\pi^i \models g$  iff  $g \in K_i$ . The proof proceeds by induction on the structure of the subformulas. Here we give the base case and the inductive step when  $g$  is either  $\neg h_1, h_1 \vee h_2, \mathbf{X}h_1$ , or  $h_1 \mathbf{U}h_2$ .

- If  $g$  is an atomic proposition, then by the definition of an atom,  $g \in K_i$  iff  $g \in L(s_i)$ .
- If  $g = \neg h_1$  then  $\pi^i \models g$  iff  $\pi^i \not\models h_1$ . By the induction hypothesis, this is true iff  $h_1 \notin K_i$ . By the definition of  $K_i$ , this guarantees that  $g \in K_i$ .
- For every  $f_1 \vee f_2 \in CL(f)$ ,  $f_1 \vee f_2 \in K_A \Leftrightarrow f_1 \in K_A \text{ or } f_2 \in K_A$
- For every  $\neg X f_1 \in CL(f)$ ,  $\neg X f_1 \in K_A \Leftrightarrow X\neg f_1 \in K_A$
- For every  $f_1 U f_2 \in CL(f)$ ,  $f_1 U f_2 \in K_A \Leftrightarrow f_2 \in K_A \text{ or } f_1, X[f_1 U f_2] \in K_A$ .

**Definition 4.** *A nontrivial strongly connected component  $C$  of the graph  $G$  is said to be self-fulfilling iff for every atom  $A$  in  $C$  and for every  $f_1 U f_2 \in K_A$  there exists an atom  $B$  in  $C$  such that  $f_2 \in K_B$ .*

**Lemma 2.** *There exists an eventuality sequence starting at an atom  $(s, K)$  iff there is a path in  $G$  from  $(s, K)$  to a self-fulfilling strongly connected component.*

( $\Rightarrow$ ) the *if* case is trivial.

( $\Leftarrow$ ), assume that there is an eventuality sequence starting at an atom  $(s, K)$ . Consider the set  $C'$  of all atoms that appear infinitely often in the this sequence.  $C'$  is a self-fulfilling strongly connected subgraph. The  $C'$  is a strongly connected component or a strongly connected subgraph of a maximal strongly connected component  $C$  of  $G$ . For the first case, it is simple. The other case, consider a subformula  $g = h_1 U h_2$  and an atom  $(s, K) \in C$  such that  $g \in K$ . If  $h_2$  is not in an atom of  $C$ , then the  $C'$  is not self-fulfilling, because the subformula  $g$  will also appear in the atoms of  $C'$ . It is a contradiction, and  $C$  is self-fulfilling.

**Corollary 1.**  *$M, s \models E f$  iff there exists an atom  $A = (s, K)$  in  $G$  such that  $f \in K$  and there exists a path in  $G$  from  $A$  to a self-fulfilling strongly connected component.*

Algorithm 1 can be used as the basis for an LTL model checking algorithm. The complexity of this algorithm is PSPACE,  $O((|S| + |R|) \cdot 2^{O(|f|)})$ .

---

### Algorithm 1. LTL Model Checking

---

**Input:**

1: Kripke structure  $M = (S, S_0, R, L)$ ; LTL formula  $f$

**Output:**

2: If  $M, s_0 \models E f$ , return YES; otherwise, return NO

3: **procedure** CONSTRUCT GRAPH( $M, f$ )

**return**  $G(At, E)$

4: **end procedure**

5:

6: **procedure**  $G2G'(G(At, E))$

**return**  $G'(At', E', L')$

7: **end procedure**

8:

9: **for all** vertex  $A \in At'$  with  $s_A \in S_0$  **do**

10:     **if**  $f \in K_A$  **then**

11:         **if** there exists a path in  $G'$  from  $A$  to  $\{B | L'(B) = 1\}$  **then**

12:             **return**  $M, s_A \models E f$

13:             **end if**

14:     **end if**

15: **end for**

**return**

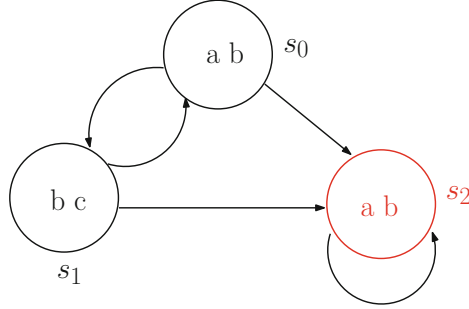
---

## 3 Model Repair of LTL with States

We propose the model repair problem of LTL. This problem asks that if given a Kripke structure  $M$  and an LTL formula is not satisfiable by the structure, what is the least atomic proposition of states that be modified for satisfiability.

**Definition 5 (modified state).** In a Kripke structure  $M = (S, S_0, R, L)$ , a state  $s \in S$  is modified if the set of atomic propositions  $L(s)$  in the state  $s$  is modified to another subset of  $2^{AP}$ .

*Example 2.* The Kripke structure  $M$  in Fig. 1 does not satisfy the LTL formula  $g = b U (-c)$ , but if the atomic propositions of the state  $s_2$  is modified for  $\{a, b\}$ , then  $M, s_0 \models Eg$  in Fig. 2.



**Fig. 2.** A modified state  $s_2$  for the example Kripke structure.

**Definition 6 (LTL model repair with states).** A LTL model repair is a tuple  $(M, \varphi)$ . Given a Kripke structure  $M = (S, S_0, R, L)$ , an LTL formula  $\varphi$ , output the least numbers of the states in  $M$  whose AP be modified such that the Kripke structure satisfy  $\varphi$ .

### 3.1 Modified LTL Graph

For the LTL model repair problem, we first construct closure  $cl(\varphi)$  of  $\varphi$ , which is the set of formulas related to the truth value of  $\varphi$ , then construct the refined LTL graph of normal atoms and modified atoms.

The closure of  $\varphi$ ,  $CL(\varphi)$  is the smallest set of formulas containing  $\varphi$  and satisfying:

- $\neg\varphi_1 \in CL(\varphi) \iff \varphi_1 \in CL(\varphi)$
- $\varphi_1 \vee \varphi_2 \in CL(\varphi) \Rightarrow \varphi_1, \varphi_2 \in CL(\varphi)$
- $X \varphi_1 \in CL(\varphi) \Rightarrow \varphi_1 \in CL(\varphi)$
- $\neg X \varphi_1 \in CL(\varphi) \Rightarrow X \neg\varphi_1 \in CL(\varphi)$
- $\varphi_1 U \varphi_2 \in CL(\varphi) \Rightarrow \varphi_1, \varphi_2, X[\varphi_1 U \varphi_2] \in CL(\varphi)$

(In the above, we identify  $\neg\neg\varphi_1$  with  $\varphi_1$ ) Notice that the size of  $CL(\varphi)$  is linear in the size of  $\varphi$ .

A normal atom is a pair  $A = (s_A, K_A)$  with  $s_A \in S$  and  $K_A \subseteq CL(\varphi) \cup AP$  such that:

- For each proposition  $p \in AP$ ,  $p \in K_A \Leftrightarrow p \in L(s_A)$
- For every  $\varphi_1 \in CL(\varphi)$ ,  $\varphi_1 \in K_A \Leftrightarrow \neg\varphi_1 \notin K_A$
- For every  $\varphi_1 \vee \varphi_2 \in CL(\varphi)$ ,  $\varphi_1 \vee \varphi_2 \in K_A \Leftrightarrow \varphi_1 \in K_A$  or  $\varphi_2 \in K_A$
- For every  $\neg X \varphi_1 \in CL(\varphi)$ ,  $\neg X \varphi_1 \in K_A \Leftrightarrow X\neg\varphi_1 \in K_A$
- For every  $\varphi_1 U \varphi_2 \in CL(\varphi)$ ,  $\varphi_1 U \varphi_2 \in K_A \Leftrightarrow \varphi_2 \in K_A$  or  $\varphi_1, X[\varphi_1 U \varphi_2] \in K_A$ .

In the LTL model repair with states problem, there are some states in which its atomic propositions should be modified for satisfying the LTL formula. Those modified states also should be displayed in the LTL constructed graph. The following modified atom is defined to show all the sets of modified states.

**Definition 7 (modified atom).** *A modified atom is also a pair  $A' = (s_A, K_A)$  with  $s_A \in S$ ,  $K_A \subseteq CL(\varphi) \cup AP$  and let  $L'(s_A)$  be a modified set of atomic propositions which is different to  $L(s_A)$  such that:*

- For each proposition  $p \in AP$ ,  $p \in K_A \Leftrightarrow p \in L'(s_A)$
- For every  $\varphi_1 \in CL(\varphi)$ ,  $\varphi_1 \in K_A \Leftrightarrow \neg\varphi_1 \notin K_A$
- For every  $\varphi_1 \vee \varphi_2 \in CL(\varphi)$ ,  $\varphi_1 \vee \varphi_2 \in K_A \Leftrightarrow \varphi_1 \in K_A$  or  $\varphi_2 \in K_A$
- For every  $\neg X \varphi_1 \in CL(\varphi)$ ,  $\neg X \varphi_1 \in K_A \Leftrightarrow X\neg\varphi_1 \in K_A$
- For every  $\varphi_1 U \varphi_2 \in CL(\varphi)$ ,  $\varphi_1 U \varphi_2 \in K_A \Leftrightarrow \varphi_2 \in K_A$  or  $\varphi_1, X[\varphi_1 U \varphi_2] \in K_A$ .

A new LTL graph  $G$  is constructed with the set of normal atoms and modified atoms as the set of vertices.  $(A, B)$  is an edge of  $G$  if only if  $(s_A, s_B) \in R$ , and for very formula  $X\varphi_1 \in CL(\varphi)$ ,  $X\varphi_1 \in K_A$  and if only if  $\varphi_1 \in K_B$ . In the new LTL graph  $G$ , the atoms is denoted as the set of normal atoms and modified atoms

**Definition 8 (modified LTL graph).** *A modified LTL graph is a tuple  $G_m(V, E, \sigma)$  constructed with a Kripke structure  $M = (S, S_0, R, L)$  and an LTL formula  $\varphi$ , where:*

- the set of vertices  $V$  is the set of normal atoms and modified atoms
- the edges  $E$  defined as:

$$(A, B) \in E \Leftrightarrow \begin{cases} (s_A, s_B) \in R; & (3) \\ \forall (X\varphi_1) \in CL(\varphi), X\varphi_1 \in K_A \Leftrightarrow \varphi_1 \in K_B & (4) \end{cases}$$

- $\sigma : V \rightarrow \{Black, Red\}$  is a color function defined as: the normal atoms are considered as black vertices and the modified ones are considered as red vertices

For the construction of the modified LTL graph, the size of normal atoms may multiplicand at most the exponential of the size of closure as  $O((|S|) \cdot 2^{O(|\varphi|)})$ , where  $|S|$  is the size of the vertices in the Kripke structure  $M$ , the size of modified atoms is the same. The size of edges of the graph  $G_m$  is about  $O((|R|) \cdot 2^{O(|\varphi|)})$ . The size of modified LTL graph  $G_m(V, E, \sigma)$  that constructed with a Kripke structure  $M = (S, S_0, R, L)$  and an LTL formula  $\varphi$  is  $|G_m| = O((|S| + |R|) \cdot 2^{O(|\varphi|)})$ .



Next, some modified version of LTL model checking are given in the following.

**Definition 9 (eventuality modified sequence).** *An eventuality modified sequence is an infinite path  $\pi$  in modified LTL graph  $G_m$  such that if  $\varphi_1 U \varphi_2 \in K_A$  for some atom  $A$  on  $\pi$ , then there exists an atom  $B$ , reachable from  $A$  along  $\pi$ , such that  $\varphi_2 \in K_B$ .*

From Lemma 1, if the LTL model repair has a solution, there exists an eventuality modified sequence starting from an atom  $(s, K)$  such that  $\varphi \in K$  in the corresponding modified LTL graph.

In the modified LTL graph, a minimal eventuality modified sequence is an eventuality sequence contained at the least read vertices in the graph.

**Definition 10 (modified LTL graph problem).** *A modified LTL path problem  $\Omega = G_m(V, E, \sigma)$  is translated from a LTL model repair problem  $(M, \varphi, k)$ , where the modified LTL graph  $G_m(V, E, \sigma)$  is constructed with the Kripke structure  $M = (S, S_0, R, L)$  and the LTL formula  $\varphi$ , output a minimal eventuality modified sequence starting at an atom  $(s_0, K)$  such that  $s_0 \in S_0, \varphi \in K$ .*

For this a minimal eventuality modified sequence, there are some atoms that appear infinitely often in this sequence. These atoms are constructed as a strongly connected graph which is a subgraph of a strongly connected component of modified LTL graph  $G$ .

**Definition 11.** *A nontrivial modified strongly connected component  $C$  of the modified LTL graph  $G_m$  is said to be self-fulfilling iff for every atom  $A$  in  $C$  and for every  $\varphi_1 U \varphi_2 \in K_A$  there exists an atom  $B$  in  $C$  such that  $\varphi_2 \in K_B$ .*

From Lemma 2, there exists an eventuality modified sequence starting from an atom  $(s, K)$  such that  $\varphi \in K$  in the modified LTL graph if only if there is a path in  $G$  from  $(s, K)$  to a self-fulfilling modified strongly connected component.

**Corollary 2.** *There exists an eventuality sequence starting at an atom  $(s, K)$  iff there is a path in  $G$  from  $(s, K)$  to a self-fulfilling strongly connected component.*

This corollary can be proofed in the same way as the Lemma 2.

**Definition 12 (optimal LTL graph).** *The optimal LTL graph problem is a tuple  $(G(V, E), \sigma, F)$ , where*

- $G(V, E)$ : is a directed LTL graph,  $V_1, V_2, \dots, V_n$  is a partition of  $V$ , and for each  $V_i, i \in [n]$ ,  $S_{i_1}, S_{i_2}, \dots, S_{i_{m_j}}$  is a partition of  $V_i$  with the same atomic propositions;
- $\sigma : S_{i_j} \rightarrow \{\text{Black}, \text{Red}\}$  assigns each vertex in the partition  $v \in S_{i_j}$  some color: the normal atoms are in Black-colored, as the modified atoms are in Red-colored;
- $F$ : is a finite set,  $f \in F$  is a element of  $F$ , for each vertex  $v_i \in V$ , there are  $A_i, B_i \subseteq F$ ;

The problem is to decide whether  $G(V, E)$  contains a path  $P$  starting at a vertex  $v_0 \in V_1$  to a circle  $C$  which satisfy  $\forall v_i \in C, \forall f \in B_i, \exists v_j \in C, f \in A_j$ , such that the  $P$  and the  $C$  intersect every  $V_i$  only one partition and contain the least number of red partitions.

In the modified LTL graph constructed in this section, those vertexes with the same  $s_A \in S$  form  $V_i$  as a partition of  $V$ . In this group, the atoms with the same atomic propositions are composed of  $S_{i_{m_j}}$  as a partition of  $V_i$ . The vertexes that constructed by normal atoms are Black-colored, as the vertexes that constructed by modified atoms are Red-colored. The finite set  $F$  is used for checking whether a circle  $C$  is self-fulfilling.

By this means, the optimal LTL graph problem is equal to the modified LTL graph problem, since the Corollary 2. These two problems are equal to the LTL model repair with states problem, for the Lemma 1.

### 3.2 Complexity of LTL Model Repair with States

In this section, we would show the algorithm of this problem. The main idea is to search a path  $\pi$  with the first vertex  $s_0 \in S_0$  that satisfy the given LTL formula with the least changed states.

Given a modified LTL graph  $G(V, E)$  that define in the Sect. 3, whether there is an eventuality sequence starting at an atom  $(s, K)$ .

$M, s \models E\varphi$  if only if there exists an atom  $A = (s, K)$  in  $G$  such that  $\varphi \in K$  and there exists a path in  $G$  from  $A$  to a self-fulfilling circle with the least number of red atoms. This optimal problem would translate into a decision version introduced in following.

**Definition 13 (k LTL graph).** The  $k$  LTL graph problem is a tuple  $((G(V, E), \sigma, F), k)$ , where

- $G(V, E)$ : is a directed graph,  $V_1, V_2, \dots, V_n$  is a partition of  $V$ , and for each  $V_i, i \in [n]$ ,  $S_{i_1}, S_{i_2}, \dots, S_{i_{m_j}}$  is a partition of  $V_i$  with the same atomic propositions;
- $\sigma : S_{i_j} \rightarrow \{\text{Black}, \text{Red}\}$  assigns each vertex in the partition  $v \in S_{i_j}$  a some color: the normal atoms are in Black-colored, as the modified atoms are in Red-colored;
- $F$ : is a finite set,  $f \in F$  is a element of  $F$ , for each vertex  $v_i \in V$ , there are  $A_i, B_i \subseteq F$ ;

The problem is to decide whether  $G(V, E)$  contains a path  $P$  starting at a vertex  $v_0 \in V_1$  to a circle  $C$  which satisfy  $\forall v_i \in C, \forall f \in B_i, \exists v_j \in C, f \in A_j$ , such that the  $P$  and the  $C$  intersect every  $V_i$  only one partition and contain at most  $k$  red partitions.

This problem is a decision problem with  $k$ , and only return the results YES or NO. The optimal LTL graph can be reduced to this problem in polynomial time.

**Lemma 3.** *A modified LTL problem is a tuple  $(M, \varphi)$ . Given a Kripke structure  $M = (S, S_0, R, L)$ , an LTL formula  $\varphi$ , there exist a algorithm to find an eventuality sequence starting at an atom  $(s_0, k)$  which contain at least red vertices, and its complexity is EXPTIME.*

*Proof.* First, we define the  $k$  decision version of the LTL model repair problem in the Definition 13. Then for each  $k$  from 1 to  $|S|$ , by using the graph combination and LTL model checking tableau algorithm mentioned in Sect. 2, there are exponential vertex combination for the modified states, each case only take  $O((|S| + |R|) \cdot 2^{O(|\varphi|)})$  time to checking.

---

**Algorithm 2.** LTL Model Repair with states

---

**Input:**

1: Kripke structure  $M = (S, S_0, R, L)$ ; LTL formula  $f$

**Output:**

2: If  $M, s_0 \models E f$ , return YES; otherwise, return k

3: **procedure** ALGORITHM 1(LTL Model Checking)  
     **return** YES or No

4:     **if** YES **then return** YES

5:     **end if**

6:     **if** No **then**

7:         **for** k **do** form 0 to  $|S|$

8:         {There exists  $C_{|S|}^k$  vertexes combination and each has at most  $|f|^k$  cases modified atomic propositions.}

9:

10:             **for** all modified atomic propositions cases **do**

11:                 **procedure** ALGORITHM 1(LTL Model Checking) **return** YES or

NO

12:                 **if** YES **then return** k

13:                 **end if**

14:             **end procedure**

15:         **end for**

16:     **end for**

17:     **end if**

18: **end procedure**

**return**

---

Algorithm 2 is used for solving the LTL Model Repair with states problem. The complexity of this algorithm is  $O(|s|^{|s|} \cdot |f|^{|s|} \cdot (|S| + |R|) \cdot 2^{|f|})$ . It is in EXPTIME.

## 4 Hardness of Decision Version

In this section, we show the hardness of decision version of LTL model repair states problem with some parameter  $k$  corresponding the number of modified states. This version problem is NP-hard even when  $k$  is fixed as a constant.

Some decision version definitions and method of LTL model repair with states are introduced in the following.

**Definition 14 (k-satisfying LTL problem).** *A k-satisfying LTL problem is a tuple  $(M, \varphi, k)$ . Given a Kripke structure  $M = (S, S_0, R, L)$ , an LTL formula  $\varphi$  and a non-negative integer  $k$ , can one modify at most  $k$  states' AP in  $M$  such that the Kripke structure satisfy  $\varphi$ .*

In the modified LTL graph, consider LTL eventuality sequence as A k-LTL path is an eventuality sequence contain at most  $k$  read vertices.

**Definition 15 (k-LTL path problem).** *A k-LTL path problem  $\Omega = (G_m(V, E, \sigma), k)$  is translated from a k-satisfying LTL problem  $(M, \varphi, k)$ , where the modified LTL graph  $G_m(V, E, \sigma)$  is constructed with the Kripke structure  $M = (S, S_0, R, L)$  and the LTL formula  $\varphi$ , whether  $G_m(V, E, \sigma)$  have a k-LTL path starting at an atom  $(s_0, K)$  such that  $s_0 \in S_0$   $\varphi \in K$ .*

The k-LTL path problem could be translated to the following graph problem.

**Definition 16 (k-directed path problem).** *The k-directed path problem is a tuple  $(G(V, E), \sigma, F, k)$ , where*

- $G(V, E)$ : is a directed graph,  $V_1, V_2, \dots, V_n$  is a partition of  $V$ , and for each  $V_i, i \in [n]$ ,  $S_{i_1}, S_{i_2}, \dots, S_{i_{m_j}}$  is a partition of  $V_i$  with the same atomic propositions;
- $\sigma : S_{i_j} \rightarrow \{\text{Black}, \text{Red}\}$  assigns each vertex in the partition  $v \in S_{i_j}$  a some color: the normal atoms are in Black-colored, as the modified atoms are in Red-colored;
- $F$ : is a finite set,  $f \in F$  is a element of  $F$ , for each vertex  $v_i \in V$ , there are  $A_i, B_i \subseteq F$ ;
- $k$ : is a non-negative integer.

*The problem is to decide whether  $G(V, E)$  contains a k-LTL path  $P$  starting at a vertex  $v_0 \in V_1$  to a circle  $C$  which satisfy  $\forall v_i \in C, \forall f \in B_i, \exists v_j \in C, f \in A_j$ , such that the  $P$  and the  $C$  intersect every  $V_i$ .*

The following problem could be considered as a simple version of the k-directed path problem, by taking the ending circle  $C$  as a vertex, then only considering the path staring from the initial vertex and ending at the circle.

**Definition 17 (path problem with paired vertexes).** *path problem with paired vertexes:*

*Input: Give a directed graph  $G = (V, E)$ , such that  $s, t \in V$ ,  $S = \{\{u_1, v_1\}, \{u_2, v_2\}, \dots\}$  with each  $u_i, v_i \in V$  and  $\{u_i, v_i\} \cap \{u_j, v_j\} = \emptyset$ , for  $i \neq j$ .*

*Goal: Whether a path  $P$  form  $s, t$ , such that for each  $i$ ,  $P$  uses at most one of  $u_i$  and  $v_i$ .*

The general problem is a simple version of the k-LTL path problem, which consider the parameterized  $k$  as a constant, an fixed positive integer.

**Definition 18 (Zero-One Equations).** *Zero-One Equations (ZOE) problem is given as follows:*

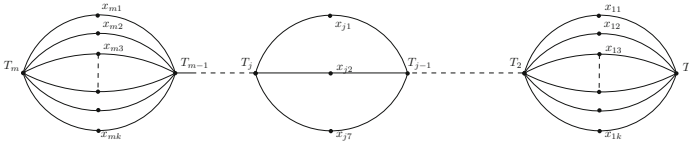
*Input: a  $m \times n$  matrix  $\mathbf{A}$ , all of whose entries are 0 or 1,*

*Goal: Find a  $n$ -vector  $X$ , all of whose entries are 0 or 1, such that  $\mathbf{A}X = \mathbf{1}$ , where  $\mathbf{1}$  denotes the all-ones vector.*

ZOE is a particularly clean special case of integer linear programming (ILP) that is very hard in and of itself: the goal is to find a vector  $\mathbf{x}$  of 0's and 1's satisfying  $\mathbf{A}\mathbf{x} = \mathbf{1}$ , where  $\mathbf{A}$  is an  $m \times n$  matrix with 0, 1 entries and  $\mathbf{1}$  is the  $m$ -vector of all 1's. It is NP-complete problem [9].

**Lemma 4.** *The path problem with paired vertexes problem is NP-hard.*

*Proof.* Now the polynomial reduction of ZOE to path problem with paired vertexes is given in the following.



**Fig. 3.** Equation parallel edges.

Given an instance of ZOE,  $\mathbf{A}\mathbf{x} = \mathbf{1}$  (where  $\mathbf{A}$  is an  $m \times n$  matrix with 0, 1 entries, and thus describes  $m$  equations in  $n$  variables), the graph we construct has the very simple structure  $s$ : a path that starts from  $s$ , contains  $m + n$  parallel edges, the first  $m$  parallel edges is correspond to  $m$  equations, and the last  $n$  edges is to  $n$  variables. For each equation  $x_{j_1} + \dots + x_{j_k} = 1$  involving  $k$  variables, we have  $k$  parallel edges with  $k$  vertex, one for every variable appearing in the equation. For example: if the  $m_j$  equation is  $x_1 + x_2 + x_7 = 1$ , this equation could be considered as shown in Fig. 3.

And for each variable, we have two parallel path (corresponding to  $x_i = 0$  and  $x_i = 1$ ). The number of vertexes in the  $x_i = 0$  path is sum of  $x_i$  in  $m$  equations plus 1, the other path contains only one vertex. For the  $m_j$  equation is  $x_1 + x_2 + x_7 = 1$ , so there are three vertexes  $\bar{x}_1^j, \bar{x}_2^j, \bar{x}_7^j$  in the three parallel edges the corresponding to  $m_j$  equation. For the variable  $x_1$ , there also a vertex  $\bar{x}_1^j$  in the path corresponding to  $x_i = 0$  as shown in Fig. 4. This paired vertexes  $\{x_{j1}, \bar{x}_1^j\}$  is in the set  $S$  that in the path problem with paired vertexes.

This is the whole reduction graph in Fig. 5. Evidently, a path from starts from  $S_0$  to  $T_m$  in this graph could traverse the  $m + n$  collections of parallel edges one by one, choosing one edge from each collection. This way, the cycle chooses for each variable a value 0 or 1 and for each equation, a variable appearing in it.

The problem of path problem with paired vertexes is the fix  $k$  simple version for the k-LTL path problem. For the reduction of ZOE to path problem with paired vertexes, it is NP-hard, even  $k$  fixed as a constant.

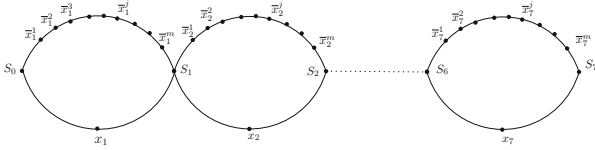


Fig. 4. Value parallel edges.

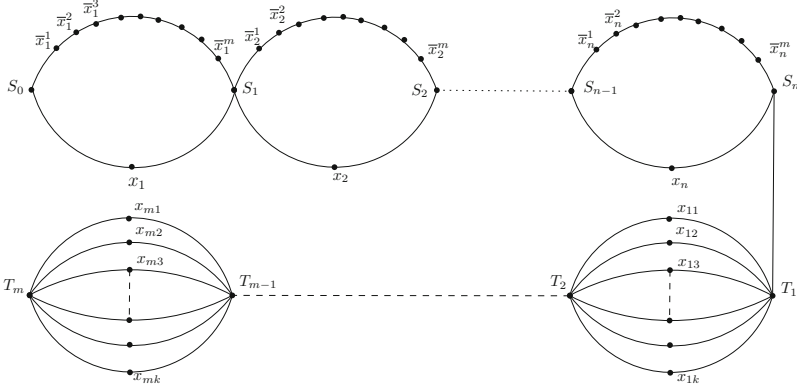


Fig. 5. Reduction graph.

**Corollary 3.** *The  $k$ -LTL path problem is NP-hard, when  $k$  is fixed as a constant.*

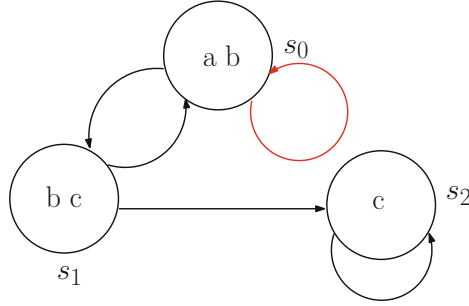
We known that the LTL model checking and satisfiability problems are PSPACE-completeness. Obviously, the model repair with states problem has a PSPACE low bound, since LTL model checking can be easily reduced to the sub problem of it. But from this corollary, it does not exist any PSPACE algorithm in which the complexity form is  $O((|S| + |R|) \cdot 2^{O(|f|)})$ , even for approximate algorithm with a finite constant approximate ratio, unless  $P = NP$ .

### 5 Model Repair with Transition Relations

In this section, another LTL model repair is defined by modifying the transition relations in the Kripke Structure for satisfying a given LTL formula. This problem is described as computing the least the number of transition relations in a Kripke structure  $M$  that should modified for satisfying a given LTL formula, when  $M$  does not satisfy the LTL formula.

**Definition 19 (modified transition relation).** *In a Kripke structure  $M = (S, S_0, R, L)$ , a transition relations is modified if the original  $R(s, s')$  has been modified for  $R(s, s'')$ , in which  $s''$  is another state in the Kripke structure.*

*Example 3.* The Kripke structure  $M$  in Fig. 6 does not satisfy  $M, s_0 \models EG(a)$ , the formula  $G(a)$ , but if the transition relation  $R(s_0, s_2)$  in the  $M$  is modified for  $R(s_0, s_0)$ , then  $M, s_0 \models EG(a)$  in Fig. 6.



**Fig. 6.** A modified transition relation for the example Kripke structure.

**Definition 20 (LTL model repair with transition relations).** A LTL model repair is a tuple  $(M, \varphi)$ . Given a Kripke structure  $M = (S, S_0, R, L)$ , an LTL formula  $\varphi$ , output the least numbers of the transition relations that be modified such that the Kripke structure satisfy  $\varphi$ .

In the similar method of Sect. 3, the LTL model repair with transition relations could be transition to the following problem.

**Definition 21 (The k-LTL model repair with transition relations problem).** The optional LTL edge problem is a tuple  $(G(V, E), \sigma, F, k)$ , where

- $G(V, E)$ : is a directed graph,  $E_1, E_2, \dots, E_n$  is a partition of  $E$ , and for each  $E_i, i \in [n]$ ,  $W_{i_1}, W_{i_2}, \dots, W_{i_{m_j}}$  is a partition of  $E_i$  with the same transition relations;
- $\sigma : S_{i_j} \rightarrow \{\text{Black}, \text{Red}\}$  assigns each edge in the partition  $e \in S_{i_j}$  a some color, the normal edges are in Black-colored, as the modified edges are in Red-colored
- $F$ : is a finite set,  $f \in F$  is a element of  $F$ , for each vertex  $v_i \in V$ , there are  $A_i, B_i \subseteq F$ ;
- $k$ : is a non-negative integer.

The problem is to decide whether  $G(V, E)$  contains a path  $P$  starting at a vertex  $v_0$  to a circle  $C$  which satisfy  $\forall v_i \in C, \forall f \in B_i, \exists v_j \in C, f \in A_j$ , such that the  $P$  and the  $C$  intersect every  $V_i$  only one partition and contain the least number of red partitions.

Using the similar algorithm, the LTL model repair with transition relations has an EXPTIME algorithm.

**Lemma 5.** The LTL model repair with transition relations problem is in EXP-TIME.

For the hardness of decision version of LTL model repair transition relations problem some parameter  $k$  corresponding the number of modified states. This problem is NP-hard, even  $k$  fixed as a constant.

**Corollary 4.** *The  $k$ -LTL model repair with transition relations problem is **NP-hard**, when  $k$  is fixed as a constant.*

## 6 Examples

In this section, we show some examples to support the applicability of our approach on real programs.

### 6.1 MinMax Example

We show a simple process to present a more general fault model, which is shown in Algorithm 3. The algorithm assigns the minimal and maximal values out of three input values to least and most [13].

The fault is located in Line 13 of Algorithm 3, where *input3* is assigned to *most* (instead of *least*), which is one of five single fault diagnoses found by a model-based debugger based on [16].

---

#### Algorithm 3. MinMax Example

---

```

1: int least = 0
2: int most = 0
3: if most < input1 then most = input1
4: end if
5: if most < input2 then most = input2
6: end if
7: if most < input3 then most = input3
8: end if
9: if least > input1 then least = input1
10: end if
11: if least > input2 then least = input2
12: end if
13: if least > input3 then most = input3
14: end if
15: return least ≤ most

```

---

### 6.2 Concurrent Program Example

The concurrent program example from [4] contains two processes  $A$  and  $B$  introduced in the Algorithms 4 and 5. Two processes  $A$  and  $B$  are described to share the control variables *flags* and *turns*, which are used to avoid concurrent access



---

**Algorithm 4.** Process A

---

```

1: flag1A :=true
2: turn1B :=false
3: if flag1B and turn1B then
4:   goto 3
5: end if
6: x := x and y
7: flag1A :=false
8: if turn1B then
9:   flag2A :=true
10:  turn2B :=true
11:  if flag2B and turn2B then
12:    goto 11
13:  end if
14:  y :=false
15:  flag2A :=false
16: end if
17: goto 1

```

---



---

**Algorithm 5.** Process B

---

```

1: flag1B :=true
2: turn1B :=false
3: if flag1A and !turn1B then
4:   goto 3
5: end if
6: x := x and y
7: flag2B :=true
8: turn2B :=false
9: if flag2A and !turn2B then
10:  goto 9
11: end if
12: y := not y
13: x := x or y
14: flag2B :=false
15: flag1B :=false
16: goto 1

```

---

to the two common boolean variables  $x$  and  $y$ . The program is executed obeying an entry and exit protocol that nondeterministically yields control to either  $A$  and  $B$ , and stores its corresponding operation.

By analyzing the programs  $A$  and  $B$ , we can find that the program is not correct, even under fair schedules. the fault is that in the Line 2 of process  $A$ , the control variable  $turn1B$  is set to *false*. This could case both a deadlock and a violation of the critical region of  $x$ . It should be  $turn1B := true$ . Even in those small examples, however, detecting the error is not immediate for the non-expert.

## 7 Conclusion

In the paper, we present an extend approach to performing model repair of linear-time temporal logic, which asks the least atomic proposition of states that be modified for satisfiability when given a Kripke structure  $M$  and an LTL formula  $\varphi$  is not satisfiable by the structure. This paper gives an algorithm to solve this problem, and the complexity of this algorithm is in EXPTIME.

For the decision version of the problem with a positive integer  $k$  for the number of modified states, we have given a reduction from ZOE problem to the fix parameter constant for the modified number to prove that it remains NP-hard even if  $k$  is a constant. This problem would be considered as the LTL satisfiability problem if given the states or transition relations of the model structure. Although LTL model checking and satisfiability problems are PSPACE-completeness. There does not exist any PSPACE algorithm in which the complexity form is  $O((|S| + |R|) \cdot 2^{O(|f|)})$  for the LTL model repair problem, even for approximate algorithm with a finite constant approximate ratio, unless  $P = NP$ .

We also define the transition relations version of LTL model repair problem, and show that it is in the same complexity.

**Acknowledgment.** The support from the National Science Foundation of China (61772336, 61732013, 61472239) and the Key R&D Project of Zhejiang Province (2017C02036) is acknowledged.

## References

1. Balbiani, P., Jean-François, C.: Computational complexity of propositional linear temporal logics based on qualitative spatial or temporal reasoning. In: Armando, A. (ed.) FroCoS 2002. LNCS (LNAI), vol. 2309, pp. 162–176. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45988-X\\_13](https://doi.org/10.1007/3-540-45988-X_13)
2. Bauland, M., Schneider, T., Schnoor, H., Schnoor, I., Vollmer, H.: The complexity of generalized satisfiability for linear temporal logic. In: Seidl, H. (ed.) FoSSaCS 2007. LNCS, vol. 4423, pp. 48–62. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-71389-0\\_5](https://doi.org/10.1007/978-3-540-71389-0_5)
3. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwegs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-48683-6\\_25](https://doi.org/10.1007/3-540-48683-6_25)
4. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing model checking in verification by AI techniques. *Artif. Intell.* **112**(1), 57–104 (1999)
5. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000). [https://doi.org/10.1007/10722167\\_34](https://doi.org/10.1007/10722167_34)
6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) *Logic of Programs* 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). <https://doi.org/10.1007/BFb0025774>

7. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). [https://doi.org/10.1007/10722167\\_15](https://doi.org/10.1007/10722167_15)
8. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, London (1999)
9. Dasgupta, S., Papadimitriou, C.H., Vazirani, U.V.: Algorithms (2016)
10. Emerson, E.A., Halpern, J.Y.: Sometimes and not never revisited: on branching versus linear time temporal logic. *J. ACM (JACM)* **33**(1), 151–178 (1986)
11. Finkbeiner, B., Torfah, H.: Counting models of linear-time temporal logic. In: Dediu, A.-H., Martín-Vide, C., Sierra-Rodríguez, J.-L., Truthe, B. (eds.) LATA 2014. LNCS, vol. 8370, pp. 360–371. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-04921-2\\_29](https://doi.org/10.1007/978-3-319-04921-2_29)
12. Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 163–173. ACM (1980)
13. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf. (STTT)* **8**(3), 229–247 (2006)
14. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press, Cambridge (2004)
15. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 97–107. ACM (1985)
16. Mateis, C., Stumptner, M., Wotawa, F.: A value-based diagnosis model for Java programs (2000)
17. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
18. Reynolds, M.: The complexity of decision problems for linear temporal logics. *J. Stud. Logic* **3**(1), 19–50 (2010)
19. Schnoebelen, P.: The complexity of temporal logic model checking. *Adv. Modal Logic* **4**(393–436), 35 (2002)
20. Xiang, J., Machida, F., Tadano, K., Yanoo, K., Sun, W., Maeno, Y.: A static analysis of dynamic fault trees with priority-and gates. In: 2013 Sixth Latin-American Symposium on Dependable Computing (LADC), pp. 58–67. IEEE (2013)



# Extending UML for Model Checking

Xinfeng Shu<sup>1</sup>(✉), Mengnan Wang<sup>1</sup>, and Xiaobing Wang<sup>2</sup>

<sup>1</sup> School of Computer Science and Technology,  
Xi'an University of Posts and Communications, Xi'an 710061, China  
shuxf@xupt.edu.cn

<sup>2</sup> Institute of Computing Theory and Technology and ISN Laboratory,  
Xidian University, Xi'an 710071, China  
xbwang@mail.xidian.edu.cn

**Abstract.** To solve the problem that UML (Unified Modeling Language) design model cannot be model-checked for its rough semantics, an extension of UML, called xUML4MC, is defined by extending the activity diagram with accurate syntax and semantics to model the activities of an object. Besides, the technique for automatic transformation of xUML4MC model into MSVL (Modeling, Simulation and Verification Language) model is also presented, which in turn can be verified with model checking tool MSV. The formalism arms the classical visual specification language UML with the power of model checking, and helps to promote the quality of software system.

**Keywords:** Unified Modeling Language · Visual specification  
System modeling · Formal verification · Model checking

## 1 Introduction

Unified Modeling Language (UML) [1], an object-oriented visual modeling language both adopted as the standard by Object Management Group (OMG) and International Organization for Standardization (ISO) [2], is widely used to visualize the analysis and design of the software system under development. UML provides a variety of diagrams (e.g., Class Diagram, Activity Diagram) to model the software system from different levels and perspectives, which can meet the modeling needs of all software development stages from requirement to deployment. However, UML is a semi-formal language, and it only can model the design of the software in a rough level, especially in describing the activities of an object. As a result, the correctness of the UML design model can hardly be automatically verified by model checking [3] approach directly.

Modeling, Simulation and Verification Language (MSVL) [4, 5] is an executable subset of Projection Temporal Logic (PTL) [6–9] with framing technique. It provides a rich set of data types (e.g., char, integer, pointer, string),

---

This research is supported by the Industrial Research Project of Shaanxi Province (No. 2017GY-076), the NSFC Grant No. 61672403.

data structures (e.g., array, list), semaphore, boolean and arithmetic expressions, as well as powerful statements [10, 11]. Further, Propositional Projection Temporal Logic (PPTL), the propositional subset of PTL, has the expressiveness power of the full regular expressions [12], which enable us to model, simulate and verify the concurrent and reactive systems within a same logical system [13]. MSVL has its specific model checking tool MSV and has been successfully used to specification and verification of typical concurrent systems [14–18].

To solve the problem of model checking UML design models, we are motivated to define a visual modeling language, called Extended UML for Model Checking (xUML4MC), by extending UML with accurate syntax and semantics to model the activities of an object. Thus, the system design model and the model checking model are unified as one xUML4MC design model, which can be created at one time by engineers when designing the system. Further, the algorithm to translate from xUML4MC model to MSVL model is formalized, which in turn can be model checked with model checking tool MSV.

The rest of the paper is organized as follows. In the next section, UML and MSVL are briefly introduced. In Sect. 3, the visual modeling language xUML4MC is defined. In Sect. 4, the algorithm to translate from xUML4MC model to MSVL model is given. In Sect. 5, related works are addressed. Finally, conclusions are drawn in Sect. 6.

## 2 Preliminaries

### 2.1 Unified Modeling Language

The Unified Modeling Language (UML) [1], a general-purpose, developmental, modeling language, was proposed by Booch, Jacobson and Rumbaugh in 1994–1995 to standardize the disparate notational systems and approaches to object-oriented software design [19]. In 1997 and 2005, UML was adopted as a standard by the OMG [2] and published by the ISO as an approved ISO standard [2] respectively.

UML provides a variety of diagrams shown in Table 1 to model the software system from different levels and perspectives. UML diagrams can be classified into static diagrams and dynamic diagrams. Static diagrams are used to represent the architecture of the software system, whereas dynamic diagrams are employed to describe the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. Besides, UML provides package diagram to manage the analysis and design models from different abstract levels while modeling complex software systems. The syntax and semantics of class diagram and activity diagram of UML are extended in xUML4MC, which are introduced in the next section.

To avoid being too complex, UML provides three kinds of extensibility mechanisms, i.e., stereotypes, tagged values and constraints, to let modeler add new building blocks, modify the properties of existing ones and even change their semantics. The stereotypes enable users to extend or create their own modeling symbols, and the stereotype name can be written in double angle brackets within

**Table 1.** The composition of UML

Major Area	Diagrams	Main Concepts
Structural	Class	Class, Association, Generalization, Dependency, Realization, Interface
	Component	Component, Interface, Dependency, Realization
	Composite structure	Component, Port, Part, CollaborationUse, Connector, Role binding
	Deployment	Node, Component, Dependency, Location
	Profile	Stereotype, Metaclass, Profile, Extension, ProfileApplication, ElementImport, PackageImport
Dynamic	State machine	State, Event, Transition, Action
	Activity	<i>Initial</i> , State, Activity, <i>Final</i> , <i>Function call</i> , <i>Activity call</i> , <i>Action</i> , Completion, Transition, Fork, Join
	Sequence	Interaction, Object, Message, Activation
	Communication	Frame, Lifeline, Message
	Interaction overview	Frame, InteractionUse, Interaction
	Timing	Frame, Message, Message label, Lifeline, GeneralOrdering
	Use case	Use case, Actor, Association, Extend, Include, Generalization
Model management	Package	Package, Subsystem, Model
Extensibility	All	Stereotypes, Tagged values, Constraint

the model element. The tagged values are used to add some extra information (e.g., developer information and code testing) in the form of key-value pairs to the model elements. The constraints are the semantic restriction on elements in the form of text expressions of other implicitly interpreted language such as mathematical symbols and OCL.

## 2.2 Modeling, Simulation and Verification Language

Modeling, Simulation and Verification Language (MSVL) is an executable subset of PTL [6]. With MSVL, expressions can be regarded as the PTL terms and statements as treated as the PTL formulas. In the following, we briefly introduce the kernel of MSVL. For more details, please refer to literatures [4, 5].

**Data Type.** MSVL provides a rich set of data types. The fundamental types include unsigned character (char), unsigned integer (int) and floating point number (float). Besides, there is a hierarchy of derived data types built with the fundamental types, including string (string), list (list), pointer (pointer), array (array), structure (struct) and union (union).

**Expression.** The arithmetic expressions  $e$  and boolean expressions  $b$  of MSVL are inductively defined as follows:

$$\begin{aligned}
 e &::= d \mid x \mid \bigcirc e \mid \ominus e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid e_1 \% e_2 \\
 b &::= true \mid false \mid \neg b \mid b_1 \wedge b_2 \mid e_1 = e_2 \mid e_1 \leq e_2
 \end{aligned}$$

where  $d$  is an integer or a floating point number;  $x \in V$  is a static or dynamic variable;  $\bigcirc e$  ( $\ominus e$ ) refers to the value of expression  $e$  at the next (previous) state.

**Statement.** The elementary statements in MSVL are defined as follows:

- (1) Immediate Assign  $x \leftarrow e \stackrel{\text{def}}{=} x = e \wedge p_x$
- (2) Unit Assignment  $x := e \stackrel{\text{def}}{=} \bigcirc x = e \wedge \bigcirc p_x \wedge skip$
- (3) Conjunction  $S_1 \text{ and } S_2 \stackrel{\text{def}}{=} S_1 \wedge S_2$
- (4) Selection  $S_1 \text{ or } S_2 \stackrel{\text{def}}{=} S_1 \vee S_2$
- (5) Next  $next S \stackrel{\text{def}}{=} \bigcirc S$
- (6) Always  $always S \stackrel{\text{def}}{=} \square S$
- (7) Termination  $empty \stackrel{\text{def}}{=} \neg \bigcirc true$
- (8) Skip  $skip \stackrel{\text{def}}{=} \bigcirc \varepsilon$
- (9) Sequential  $S_1; S_2 \stackrel{\text{def}}{=} (S_1, S_2)prj \varepsilon$
- (10) Local  $exist x : S \stackrel{\text{def}}{=} \exists x : S$
- (11) State Frame  $lbf(x) \stackrel{\text{def}}{=} \neg af(x) \rightarrow \exists b : (\bigcirc x = b \wedge x \stackrel{\text{def}}{=} b)$
- (12) Interval Frame  $frame(x) \stackrel{\text{def}}{=} \square(\bar{\varepsilon} \rightarrow \bigcirc(lbf(x)))$
- (13) Projection  $(S_1, \dots, S_m)prj S$
- (14) Condition  $\text{if } b \text{ then } S_1 \text{ else } S_2 \stackrel{\text{def}}{=} (b \rightarrow S_1) \wedge (\neg b \rightarrow S_2)$
- (15) While  $\text{while } b \text{ do } S \stackrel{\text{def}}{=} (b \wedge S)^* \wedge \square(\varepsilon \rightarrow \neg b)$
- (16) Await  $await(b) \stackrel{\text{def}}{=} \bigwedge_{x \in V_b} frame(x) \wedge \square(\varepsilon \leftrightarrow b)$
- (17) Parallel  $S_1 || S_2 \stackrel{\text{def}}{=} ((S_1; true) \wedge S_2) \vee (S_1 \wedge (S_2; true))$   
 $\vee (S_1 \wedge S_2)$

where  $x$  is a variable,  $e$  is an arbitrary expression,  $b$  is a boolean expression, and  $S_1, \dots, S_m, S$  are all MSVL statements. The immediate assignment  $x \leftarrow e$ , unit assignment  $x := e$ ,  $empty$ ,  $lbf(x)$  and  $frame(x)$  are basic statements, and the left composite ones.

### 3 Structures of xUML4MC

The language xUML4MC is an extension of UML by introducing accurate syntax and semantics to model the activities of an object. As shown in the rows with gray background in Table 1, the extensions are made to the class diagram and activity diagram, and the extension points are marked as the bold italic font.

Except for keeping the other elements and diagrams of UML unchanged, the extension in xUML4MC consists of fundamental notations and visual notations. The former are used to describe the attributes and the details of activities of the objects to be modeled, and the latter are used in class diagram and activity diagram to model the object construction of the software and activities of each object respectively.

### 3.1 Fundamental Notations

The fundamental notations of xUML4MC include data types, expressions as well as elementary statements. The data types and expressions are mainly defined with Object Constraint Language (OCL) [20].

**Definition 1 (Data Type).** The data types of xUML4MC consist of primitive types and collections. The primitive types include boolean (*Boolean*), integer (*Integer*), floating point number (*Real*) and string (*String*). The collection types include set (*Set(type)*), sequence (*Sequence(type)*) and array (*arrayname[len] : type*), where *type* can be any xUML4MC types or user-defined classes.

**Definition 2 (Expression).** Let  $d$  be a constant,  $x$  be a variable respectively, and  $obj$  be an object. The arithmetic expressions  $e$  and boolean expressions  $b$  of xUML4MC are inductively defined as follows:

$$\begin{aligned} e &::= d \mid x \mid obj.attr \mid e_1 \ op_1 \ e_2 \ (op_1 ::= + \mid - \mid * \mid /) \\ b &::= true \mid false \mid e_1 \ op_2 \ e_2 \ (op_2 ::= < \mid > \mid >= \mid <= \mid = \mid <>) \\ &\quad b_1 \ op_3 \ b_2 \ (op_3 ::= and \mid or \mid xor) \end{aligned}$$

where  $e_1, e_2, \dots, e_n$  are well-formed expressions;  $op_1, op_2$  and  $op_3$  denote the traditional arithmetic operator, relational operator and logical operator respectively; *attr* (*fun*) refers to an attribute (a member function) of object  $obj$ .

**Definition 3 (Elementary Statement).** Let *type* be a data type,  $x$  be a variable and  $obj$  be an object. The elementary statement  $s$  of xUML4MC are inductively defined as follows:

$$\begin{aligned} s &::= x : type \mid x : type := e \mid x := e \mid x := obj.fun(e_1, \dots, e_n) \\ &\quad \mid obj : Class(e_1, \dots, e_n) \mid obj.fun(e_1, \dots, e_n) \mid s_1 \ and \ s_2 \end{aligned}$$

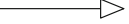


where  $e, e_1, \dots, e_n$  ( $n \geq 0$ ) are expressions; *attr* refers an attribute of object  $obj$ ; *fun* denotes a member function of object  $obj$  with  $n$  parameters;  $s_1$  and  $s_2$  are both well-formed statements.  $s_1$  and  $s_2$  is called *composed statement*, and the others are called *basic statements*. Intuitively, composed statement  $s_1$  and  $s_2$  means statements  $s_1$  and  $s_2$  execute at the same time.



### 3.2 Visual Notations

The visual notations for class diagram and activity diagram of xUML4MC are defined in Tables 2 and 3 respectively. The shapes used in class diagram, i.e., class, generalization, association and composition, are identical to that of UML. Besides, the definitions of attributes and functions in class notation of UML are also kept unchanged in xUML4MC.



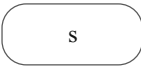

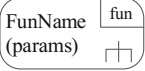
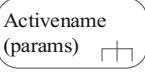
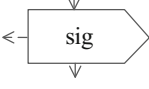
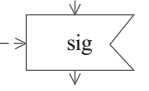


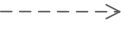
**Table 2.** Visual notations for class diagram

Element	Name	Description			
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center;">Name</td> </tr> <tr> <td style="text-align: center;">Attribute list</td> </tr> <tr> <td style="text-align: center;">Function list</td> </tr> </table>	Name	Attribute list	Function list	CLASS	Represents a class which consists of class name, attributes and member functions. The static attributes and function are marked with underline. Beside, the entry function of the UML model is identified with double underlines.
Name					
Attribute list					
Function list					
	Generalization	Denotes the generalization relationship between the subclass and super class, which triangle shape is on the super class end and the line connects to the subclass.			
	Association	Represents the static relationship shared among the objects of two classes. Associations can be unidirectional or bidirectional, but must have multiplicity decorations.			
	Composition	Represents the whole-part relationship between two classes, which diamond shape is on the containing class end and the line connects to the contained class.			

The major visual notations used in activity diagram keep identical to that in UML. Besides, we also make some necessary extensions for accurately modeling the activities of an object:

- Attach the information of the corresponding function header to the initial state, including function name, parameters and return type. For any activity diagram, the initial state is unique which represents the entry of the diagram.
- Attach the return value of the activity to the final state. A activity diagram may have many final states.
- Arm the action node with the elementary statements defined in Definition 3. Note that the basic statements in an action node execute at the same time, so they cannot exist contradiction. For example, the statement “ $x := 1$  and  $x := x + 1$ ” does not allowed.
- Add a new function call symbol to represent calling a function which is modeled in another separated activity. The function call symbol must be associated with the *Object*.
- Introduce a new reference symbol to represent a complicated processing step which is detailed in another separate activity diagram.

**Table 3.** Visual notations for activity diagram

Element	Name	Description
 fun(params):type	Initial State	Represents the entrance of the activity model. It includes function name, parameters and return type.
 return e	Final State	Represents the exit of an activity diagram. It may include a return value <i>e</i> of the activity diagram if it has.
 s	Action Node	Indicates some elementary statements are performed at the same time, e.g., "x:Integer := 1", "x:=1 and y:=2".
 exp	Decision	Denotes where a decision is necessary. The decision is described as a boolean expression <i>exp</i> in the symbol.
	Function Call	Indicates a function being invoked and the function is modeled in a separate activity diagram. It contains the name of the function and the possible arguments passed to the function.
	Activity Reference	Represents a reference to another activity. It contains the name of the activity and the arguments passed to the activity.
	Signal Send	Denotes sending an asynchronous signal <i>sig</i> without waiting for a receipt.
	Signal Accept	Denotes accepting the signal <i>sig</i> sent by an object. It will keep on waiting until the signal arrives.
	Synchronization	Shows the concurrent flows. Note that all concurrent flows start at a synchronization symbol and end at another synchronization symbol.
	Control flow	Represents the control flow passing from one symbol to another.
	Object flow	Represents the dependency between a symbol and an object. It emphasizes the effect of the activity on the object.

### 3.3 System Modeling with xUML4MC

The general system modeling strategy with xUML4MC is similar to that with UML in software system design. Besides, xUML4MC employs the extended activity diagrams to describe the detailed design of each member functions of the classes in the class diagrams. What's more, a static member function of a class must be explicitly declared as the execution entry of the xUML4MC model. All these diagrams are combined together to show the system design of the software system.

In the following, we give an example of Car Sell System (CSS) to show how to model with xUML4MC. Since the transformation of the xUML4MC model into MSVL model only relies the class diagram and the activity diagrams of each member functions of the classes, the other design diagrams are skipped here. As depicted in Fig. 1, the class diagram of the CSS consists of five classes, among which *Vehicle* is the super class of *Car*, and *CarShop* has two types of *Car*, and the member function *main()* of class *App* is marked as the entry of the model.

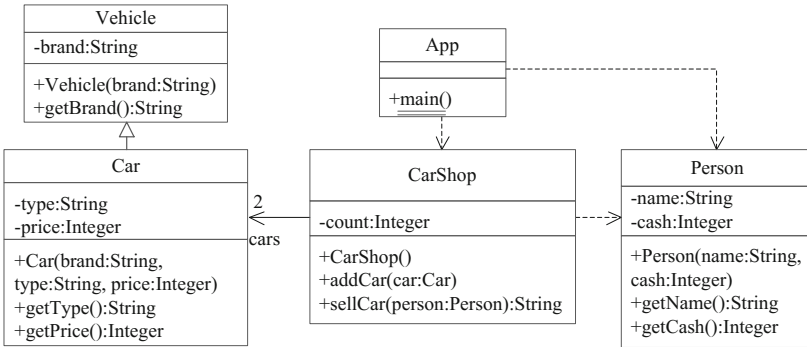


Fig. 1. Class diagram of CSS

For each member function of the class diagram of CSS, we need to create an activity diagram describing its processing logic. However, the constructor and “get” methods of each class are rather simple, we only give the activity diagrams of functions *main* and *sellCar* of classes *App* and *CarShop* respectively. The activity diagram of function *main* is shown in Fig. 2(a), within which objects *car1*, *car2* and *carshop* as well as *person* are created firstly; then objects *car1* and *car2* are added to the object *carshop*; finally, object *person* is used as the only parameter to call the member function *sellCar* of object *carshop*. The activity diagram of *sellCar* of class *CarShop* is depicted in Fig. 2(b), which compares each car’s price in object *carshop* with the *person*’s cash and returns the best-fit car’s information.

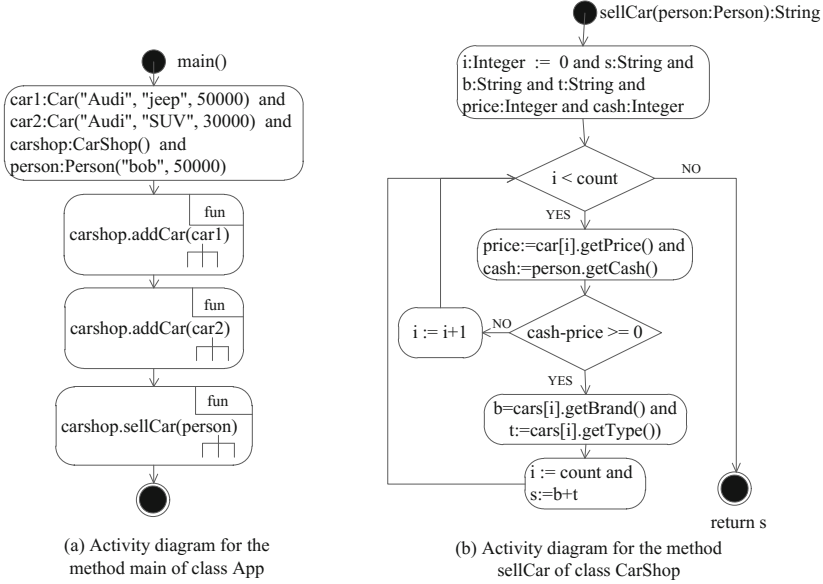


Fig. 2. Activity diagrams of CCS

## 4 Translation of xUML4MC Model into MSVL Model

In case of the xUML4MC system model being created, the left work is to automatically translate it into the MSVL model. To this end, a formal definition of xUML4MC model is given, and a abstract syntax tree (AST) for describing the syntax of the xUML4MC model is introduced. Besides, the algorithms for transforming xUML4MC model to AST and AST to MSVL model are also formalized respectively.

### 4.1 Formal Definition of xUML4MC Model

Although the xUML4MC model consists of many design diagrams of different kinds, we only give the formal definitions of class diagram and activity diagram needed for translation it into MSVL model.

**Definition 4 (Class Diagram).** The attribute *Attr*, parameter *Param*, function *Fun*, class node *CNode*, arc *CArc* among class nodes, and class diagram *CD* are defined inductively as follows:

- Attr* ::= <visibility, type, name, isStatic, initValue>
- Param* ::= <type, name>
- Fun* ::= <visibility, type, name, paramList, isStatic, isEntry>
- CNode* ::= <name, attrSet, funSet>
- CArc* ::= <arcType, nodeFrom, nodeTo, mulFrom, mulTo, roleFrom, roleTo>
- CD* ::= <CNodeSet, CArcSet>

where *visibility* denotes the visibility of an attribute or a function, and it takes the value of VIS\_PUB (public), VIS\_PRI (private) or VIS\_PRO (protected); *type* is the data type supported by xUML4MC; *isStatic* denotes whether the attribute or function is static or not (1 for static and 0 for non-static); *isEntry* equals 1 represents the corresponding function is the execution entry of xUML4MC model, otherwise it takes the value of 0; *arcType* represents the relationship type among class nodes, the value of which ranges among ARC\_GEN (generalization), ARC\_ASS (association), ARC\_COM (composition) and ARC\_DEP (dependency). The definition of *CArc* indicates an arc departs from class node *nodeFrom* and enters into *nodeTo* with the role names *roleFrom* and *roleTo*, and the corresponding multiplicities are *mulFrom* and *mulTo* respectively.

**Definition 5 (Activity Diagram).** An activity diagram *AD* can be regarded as a directed graph, and the node notation *ANode* and arc *AArc* as well as the diagram *AD* are defined inductively as follows:

$$\begin{aligned}
 ANode &::= \langle nodeType, content \rangle \\
 AArc &::= \langle arcType, nodeFrom, label, nodeTo \rangle \\
 AD &::= \langle className, ANodeSet, AArcSet, entryNode \rangle
 \end{aligned}$$

where *nodeType* denotes the type of the node notation and its values is given in Table 4; *content* is an expression or an elementary statement contained in the notation; *arcType* denotes the type of the flow line which can only take the value of ARC\_CTR (control flow) or ARC\_OBJ (object flow); *nodeFrom* and *nodeTo* represent the coming from and ending at node notations of flow line respectively; *label* is the label on the control flow and it can only take the value of “YES”/“NO” in case of the flow line departing from a decision shape; *className* shows the name of the class which the activity diagram belongs to, and *entryNode* gives the entrance node of the activity diagram.

**Table 4.** The value of node notation type [21]

Notation name	Value	Notation name	Value
Initial state	NT_ET	Final state	NT_EX
Action node	NT_PR	Function call	NT_FC
Activity reference	NT_RF	Decision	NT_DC
Synchronization	NT_FJ		

**Definition 6 (xUML4MC Model).** An xUML4MC model *xUML4MC\_Model* is defined as the pair  $xUML4MC\_Model ::= \langle CDSet, ADSet \rangle$ , where *CDSet* and *ADSet* are the sets of class diagrams and activity diagrams respectively.

### 4.2 Conversion from xUML4MC Model to Abstract Syntax Tree

On the whole, the execution of an xUML4MC model is in fact sequentially traversing each activity diagrams from the entrance to the exit, but it involves accessing the attributes of the related objects as well as handling the non-sequential structures such as branch, loop and parallel. So, we cannot transform these structures into MSVL code according to the traverse sequence directly. To solve the problem, we introduce a data structure, named Abstract Syntax Tree (AST), to analyze the syntax of an xUML4MC model.

**Abstract Syntax Tree.** The strategy of AST representing the syntax of xUML4MC model can be depicted as the figure in Fig. 3, all the classes are organized in a linked list and each class corresponds to a list node. A class node mainly consists of four data items, of which *parent* and *next* point to the next class node and parent class node respectively; *firstAttr* points to the attribute list of the class. Since the visual notations in the activity diagram are identical to that in the vMSVL [21], so the technique of Hierarchical Syntax Chart (HSC) in [21] is introduced to describe the syntax of each activity diagrams of the member functions, and the data item *firstFunc* of a class node points to the first HSC node of the class.

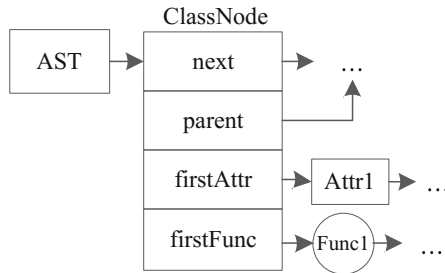


Fig. 3. Structure of abstract syntax tree

The structure of HSC can be depicted as the figure in Fig. 4. In first level, the HSC is the sequence of compound statements of functions, and the function body, a compound statement, is the sequence of statements in the function body. If the compound statement includes if, while or parallel statements, their corresponding execution breaches are also organized the sequence of compound statements, e.g., the *if* statement in the body of function *Fun1*.

According to the above analysis, the data structure of AST is defined in pseudo C Language as follows:

```

/*type of the Attribute*/
typedef struct Attribute{
    DataType type;
    string name;
    string value;
}
    
```

```

    struct Attribute *next;
}Attr;

/*type for list of ClassNode*/
typedef struct ClassNode{
    string name;
    struct ClassNode *parent;
    struct Attr *firstAttr;
    HSC firstFunc;
    struct ClassNode *next;
}CNode, * AST;
    
```

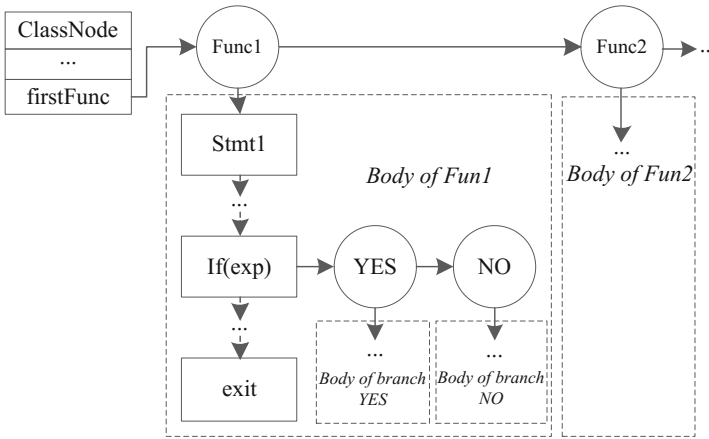


Fig. 4. Structure of hierarchical syntax chart

In structure *ClassNode*, the member *parent* points to its parent class and it equals NULL if no parent class exists; member *firstFunc* refers to the HSC of the functions of the class. The definition of HSC can be found in [21].

For a given xUML4MC model, the algorithm for constructing its corresponding AST consists of functions `xUML4MCtoAST`, `Creat_CN`, `handle_CN`, `handle_AD`, `FC2ComStmt`, and their relationships are shown in Fig. 5. Function `xUML4MCtoAST` is the entry of the algorithm, which calls functions `Creat_CN`, `handle_CN` and `handle_AD` in sequence. Function `Creat_CN` traverses the class diagrams, and for each class node in the *CNodeSet*, it creates a *CN* node and arranges them in the AST. For each *CN* node in the AST, function `handle_CN` first handles the attributes of the class as well as the relationships between *CN* and other classes. Function `handle_AD` is employed to deal with the activity diagrams of the member functions. For each member function of a *CN* node, it calls function `FC2ComStmt` to transform it into a HSC and add the HSC to the tail of *CN*'s HSC list. The code of function `FC2ComStmt` can be found in literature [21]; the other functions are relative simple, and hence their code is omitted here.

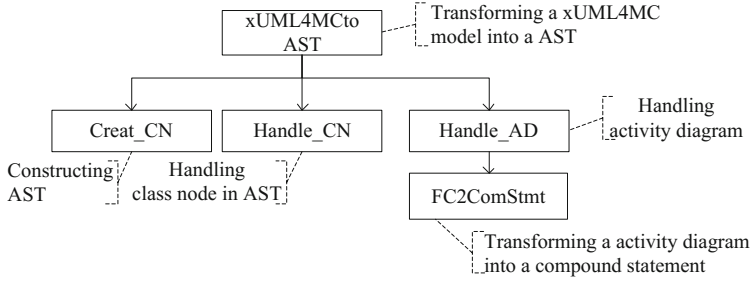


Fig. 5. Algorithm for constructing AST

### 4.3 Conversion from AST to MSVL

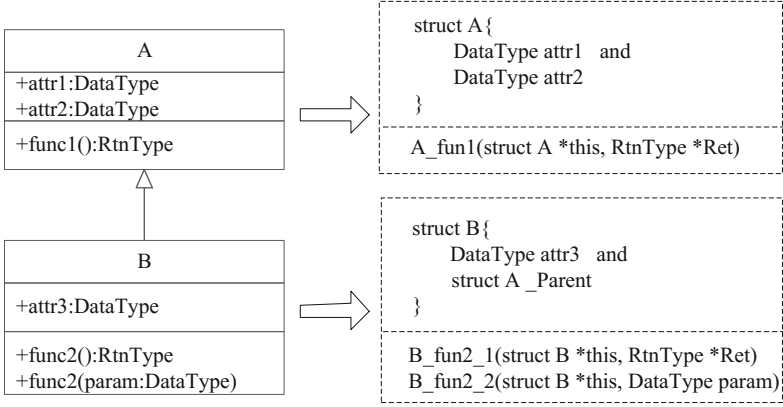
Since xUML4MC is an object-oriented modeling language whereas MSVL is a process-oriented one, we need to decompose the inheritance and encapsulation of xUML4MC in AST before transformation it into MSVL.

**Preprocessing AST.** The basic strategy of preprocessing AST can be depicted as the figure in Fig. 6. For each class in the AST (e.g., class  $A$ ), we define a struct in MSVL with the same name as the class (w.r.t. struct  $A$ ), and the members of the struct are identical to the attributes of the class. For each member function of a class, we define a MSVL function named by the original function name prefixed with the class name. Besides, for convenience of accessing the members the MSVL struct in the function body, we insert a new parameter *this* with the type of the MSVL struct to the head of the parameter list of the function. If the function has a return value, we add another new parameter named *Ret* to the tail of the parameter list. For example, corresponding to the function  $func1$  of class  $A$ , a MSVL function  $A\_func1$  is defined with two new parameters *this* and *Ret* added to the head and tail of the parameter list respectively. For a class having a parent class (e.g., class  $B$ ), we add a new member  $\_Parent$  with type of its parent class’s struct to its MSVL struct (w.r.t struct  $B$ ). For the overload of member functions (e.g., the two functions  $func2$  in class  $B$ ), we rename the functions in MSVL with the original name suffixed with the index number (w.r.t.  $B\_func2\_1$  and  $B\_func2\_2$ ).

The detailed rules for preprocessing AST are as follows:

- R1. For each class node in the AST, add value of the class name with the prefix “*struct*”.
- R2. For the attributes with initial values of a class, add the corresponding assignment statements to the beginning of the HSC of the class’s constructor, and remove the initial values from the attribute nodes.
- R3. For a class having a parent class  $par$ , add a new attribute node with the name “ $\_Parent$ ” and type of  $par$ ’s struct to its attributes list.

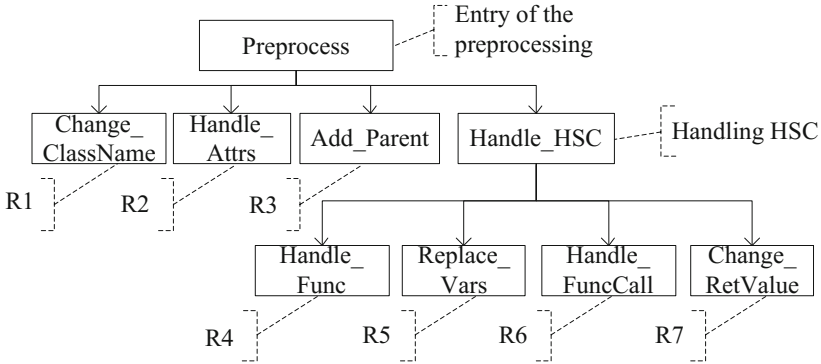




**Fig. 6.** Strategies for preprocessing AST

- R4. For each initial node in the HSC of a class *cls*, add to the function name with the prefix of *cls*' name and “\_”, and add a new parameter with the name “*this*” and type of *cls*' struct to the head of the function's parameter list. If the function has a return value of type *rtnType*, add a new parameter with the name “*Ret*” and type *rtnType* to the tail of the function's parameter list. Besides, for the functions with the same name in a class, add to the name of each overloaded functions with the suffix “\_” and index number.
- R5. For any statement *stmt* accessing an attribute *attr* of class *cls* in the *cls*' HSC, replace all the occurrence of “*attr*” with the string “*this* →” connected with the result of function call *find\_attr(cls, attr)*.
- R6. For any function call statement  $x := obj.fun(e_1, \dots, e_n)$  (w.r.t. *obj.fun* ( $e_1, \dots, e_n$ )) in the HSC and *obj* is an instance of class *cls*, replace the function call statement with the result of function call *find\_func(cls, “obj”, “fun”, paramTypeList)* connected with the string “ $e_1, \dots, e_n, \&x$ ” (w.r.t. “ $e_1, \dots, e_n$ ”), where *paramTypeList* is the data type list of the parameters  $e_1, \dots, e_n$ .
- R7. For each final state node in a HSC, if it contains a return statement *return e*, then replace the statement with  $*Ret := e$ .

According to the above rules, the algorithm for preprocessing AST consists of 9 functions whose relationships are shown in Fig. 7. Function Preprocess is the entry of the algorithm, and it traverse all class nodes in the AST. For each class node *CN*, function Preprocess first calls functions Change.ClassName, Handle\_Attrs and Add\_Parent to rename the class (Rule R1), remove the init values of attributes (Rule R2) and deal with the inherited attributes (Rule R3) respectively, and then calls function Handle\_HSC to process the HSC of *CN*. For each member function's HSC, function Handle\_HSC calls functions *Handle\_Func*, *Replace\_Vars*, *Handle\_FuncCall* and *Change\_RetValue* in sequence to handle



**Fig. 7.** Algorithm for preprocessing AST

the function header (Rule R4), replace the access of class' attributes with members of MSVL struct (Rule R5), handle the function (Rule R6) as well as deal with the return statement (Rule R7) respectively. The code of the functions are skipped here.

In case of an AST having been preprocessed, the algorithm AST2MSVL for transforming it into MSVL code is defined as follows, where function TypeConvert is used to convert the xUML4MC data type into the MSVL data type according to translation rules in Table 5. For each class node in the AST, algorithm AST2MSVL first creates a MSVL struct with the name and the attributes of the class, and then calls the function HSC2MSVL to transform the HSC of the class' member functions into MSVL code. The code for function TypeConvert is trivial and hence omitted here, and the code of function HSC2MSVL can be found in literature [21].

```

AST2MSVL(AST *ast){
    string msvlCode;
    struct ClassNode *CN = ast->firstCN;
    while(temp != NULL){
        struct Attr *attr = CN->firstAttr->next;
        string s = CN->name+"{"
        while(attr != NULL){
            string type = TypeConvert(attr->datatype);
            s = s + type + " " + attr->name + " and ";
            attr = attr->next;
        }
        s = s+"}";
        msvlCode = msvlCode + s;
        msvlCode = msvlCode + HSC2MSVL(CN->firstFunc->next);
    }
    return msvlCode;
}
  
```

**Table 5.** Translation rules for data types between xUML4MC and MSVL

xUML4MC	MSVL	xUML4MC	MSVL
Boolean	bool	Set	list
Integer	int	Sequence	list
Real	float	Array	array
String	string		

#### 4.4 Translation Example

For the xUML4MC model of CSS in Subject. 3.3, we first convert it into AST with algorithm xUML4MCtoAST, and the result is depicted in Fig. 8, where we only give the class node of class *Car* (Fig. 8(a)) and the HSC of member function *sellCar* of class *CarShop* (Fig. 8(b)) in details. The class node *Car* has two attributes (i.e., *type* and *price*), three functions (i.e., *Car*, *getType* and *getPrice*) and a parent class. The HSC of function *sellCar* has a “loop” structure controlled by the attribute *count* of class *CarShop* and variable *i*, within which an “if” structure is identified to choose an appropriate car by the price of the car and the cash of the person.

Subsequently, we employ algorithm Preprocess to handle the AST according to the 7 preprocessing rules mentioned above. The changes of the AST nodes’ content after preprocessing are attached as the text in the dashed boxes in Fig. 8. For instance, the name of class node *Car* is replaced with *struct Car*, the attributes inherited from class *Vehicle* are represented by adding a new attribute node *\_Parent* of the type *struct Vehicle*, and the header of function *sellCar* is replaced with *CarShop\_sellCar(\*this : struct CarShop, person : Struct Person, \*Ret : String)*.

Finally, we use algorithm AST2MSVL to transform the preprocessed AST into MSVL model. For save space, here we only give the MSVL code for function *sellCar* and classes *Car* and *Carshop*.

```

struct Car{
    string type and
    int price and
    struct Vehicle _Parent
};
struct Carshop{
    struct Car cars [2]
};
CarShop_sellCar(struct CarShop *this, struct Person person
, string *Ret){
    frame(i,s, b, t,price, cash) and (
        int i and i<==0 and empty;
        string s and string b and string t and int price
        and int cash;
        while(i<this.count){

```

```

Car_getPrice ( this -> cars [ i ], & price );
Person_getCash ( person , & cash );
if ( cash - price >= 0 ) {
    Veche_getBrand ( this -> cars [ i ] . _Parent , & b );
    Car_getType ( this -> cars [ i ], & t );
    i := this -> count ;
    s = b + t
} else {
    i := i + 1
}
};
*Ret := s
)
}

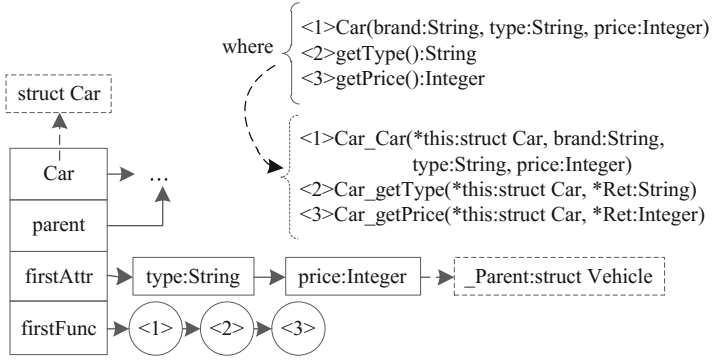
```

## 5 Related Works

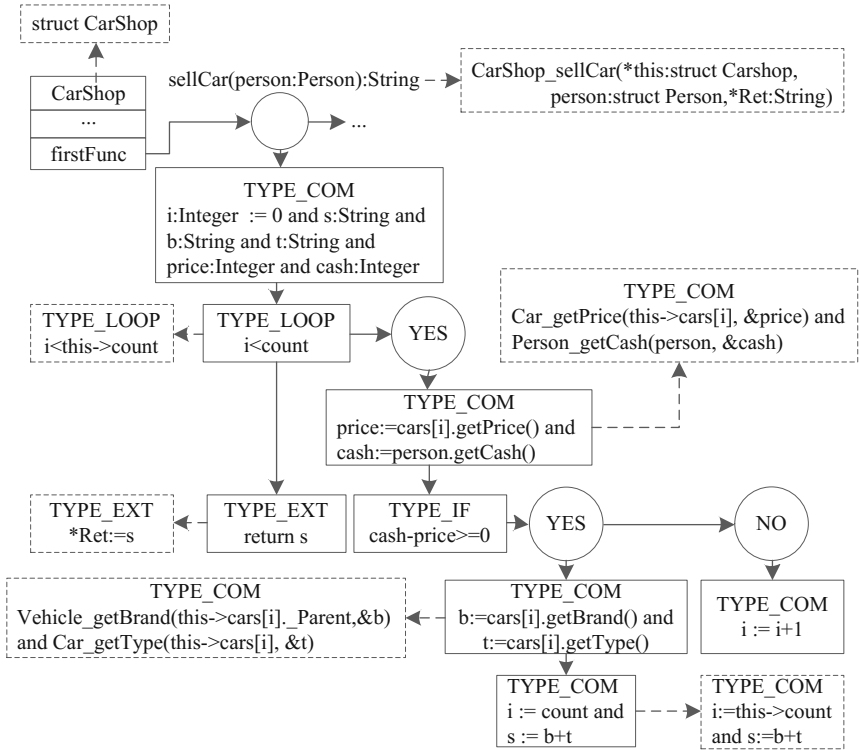
So far, there already exist some literatures [22–25] on the field of model checking UML models including class diagrams, sequence diagrams, state machine diagrams or their compositions. The main research works focus on model checking UML model with classic tools such as SPIN or NuSMV, and the basic idea is to transform the UML model into the specific modeling language (e.g., PROMEL, SMV) model w.r.t. the model checking tool. Literature [22] translates both the high-level and low-level sequence diagrams into SMV specifications and Linear Temporal Logic (LTL) based constraint, and employs NuSMV to verify the containment relationship between sequence diagrams. Literature [23] creates the PROMELA-based model from the interactions expressed in sequence diagrams, and uses SPIN model checker to simulate the execution and to verify the execution state of an interaction written in LTL formula.

Some researchers also develop the specific tools to model checking UML models. Gnesi and Mazzanti [24] present an “on the fly” model checker UMC (UML on the fly Model Checker) to verify the conformance of state machine diagrams. The tool employs labeled transition systems as the system modeling language, while the properties to be verified are specified in  $\mu$ -ACTL, an extension of the action based branching time temporal logic. Mullins and Oarga [25] present a verification tool SOCLe that offers dynamic verification of extended OCL constraints on UML models. SOCLe first translates the UML model into an Abstract State Machine by the UML compiler, and then transform it into an abstract structure called UML-valued OO Transition System ( $OOT_{S_{UML}}$ ) by ASM simulator, finally verifies the OCL constraints with the on-the-fly model checker EOCL.

Compared to the exists works, our approach extends UML with strict syntax and semantics to model the activities of the objects in the software system, and automatically transforms the model into MSVL model for further model checking. With the full system model in MSVL, one can model checking any expected properties.



(a) AST for class Car of CSS



(b) HSC for sellCar of the class CarShop

Fig. 8. The corresponding AST of the xUML4MC model of CSS

## 6 Conclusion

In this paper, we present a visual modeling language xUML4MC by extending the widely used modeling language UML with accurate syntax and semantics, and formalize the algorithms to automatically translate xUML4MC model to MSVL model for model checking. The introduction of language xUML4MC enables the software to be verified with model checking approach at the design time, which helps to promote the quality of software as well as popularize of the model checking technology in industry. In the near future, we will promote the formalism to deal with the override of inherited methods. Besides, we will develop a visual modeling tool based on the xUML4MC and apply the method to verify some typical object-oriented software system, such as web service and cloud computing.

## References

1. Object Management Group: OMG Unified Modeling Language (OMG UML), Version 2.5 (2015). <http://www.omg.org/spec/UML/2.5>
2. ISO/IEC 19501:2005 - Information Technology - Open Distributed Processing - Unified Modeling Language (UML) Version 1.4.2, Iso.org., 01 April 2005. Accessed 07 May 2015
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. In: Nato Advanced Study Institute on Deductive Program Design, pp. 305–349 (1996)
4. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. *Sci. Comput. Program.* **70**(1), 31–61 (2008)
5. Duan, Z., Koutny, M.: A framed temporal logic programming language. *J. Comput. Sci. Technol.* **19**, 333–344 (2004)
6. Duan, Z.: Temporal logic and temporal logic programming. Science Press, Beijing (2005)
7. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. *Acta Inf.* **45**(1), 43–78 (2008)
8. Duan, Z., Tian, C.: A practical decision procedure for propositional projection temporal logic with infinite models. *Theoret. Comput. Sci.* **554**, 169–190 (2014)
9. Tian, C., Duan, Z., Zhang, N.: An efficient approach for abstraction-refinement in model checking. *Theoret. Comput. Sci.* **461**, 76–85 (2012)
10. Wang, X., Tian, C., Duan, Z., Zhao, L.: MSVL: a typed language for temporal logic programming. *Front. Comput. Sci.* (2017). <https://doi.org/10.1007/s11704-016-6059-4>
11. Shu, X., Duan, Z.: Extending MSVL with semaphore. In: Dinh, T.N., Thai, M.T. (eds.) COCOON 2016. LNCS, vol. 9797, pp. 599–610. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-42634-1\\_48](https://doi.org/10.1007/978-3-319-42634-1_48)
12. Tian, C., Duan, Z.: Expressiveness of propositional projection temporal logic with star. *Theoret. Comput. Sci.* **412**(18), 1729–1744 (2011)
13. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-88194-0\\_12](https://doi.org/10.1007/978-3-540-88194-0_12)

14. Duan, Z.: Modeling and Analysis of Hybrid Systems. Science Press, Beijing (2004). <https://doi.org/10.1007/3-540-45426-8>
15. Wang, M., Duan, Z., Tian, C.: Simulation and verification of the virtual memory management system with MSVL. CSCWD 2014, pp. 360–365 (2014)
16. Yu, Y., Duan, Z., Tian, C., Yang, M.: Model checking C programs with MSVL. In: Liu, S. (ed.) SOFL 2012. LNCS, vol. 7787, pp. 87–103. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-39277-1\\_7](https://doi.org/10.1007/978-3-642-39277-1_7)
17. Shu, X., Duan, Z.: Model checking process scheduling over multi-core computer system with MSVL. In: Liu, S., Duan, Z. (eds.) SOFL+MSVL 2015. LNCS, vol. 9559, pp. 103–117. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-31220-0\\_8](https://doi.org/10.1007/978-3-319-31220-0_8)
18. Zhang, N., Duan, Z., Tian, C.: A cylinder computation model for many-core parallel computing. Theoret. Comput. Sci. **497**, 68–83 (2013)
19. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. China Machine Press, Beijing (2006)
20. Object Management Group (OMG): Object Constraint Language OMG Available Specification, Version 2.0, May 2006
21. Shu, X., Li, C., Liu, C.: A visual modeling language for MSVL. In: Liu, S., Duan, Z., Tian, C., Nagoya, F. (eds.) SOFL+MSVL 2016. LNCS, vol. 10189, pp. 220–237. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-57708-1\\_13](https://doi.org/10.1007/978-3-319-57708-1_13)
22. Muram, F.U., Tran, H., Zdun, U.: A model checking based approach for containment checking of UML sequence diagrams. In: Software Engineering Conference, pp. 73–80. IEEE (2017)
23. Lima, V., Talhi, C., Mouheb, D., et al.: Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. Electron. Notes Theoret. Comput. Sci. **254**, 143–160 (2009)
24. Gnesi, S., Mazzanti, F.: On the fly model checking of communicating UML State Machines. Acis IEEE, vol. 144, pp. 331–338 (2004)
25. Mullins, J., Oarga, R.: Model checking of extended OCL constraints on UML models in SOCLe. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 59–75. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-72952-5\\_4](https://doi.org/10.1007/978-3-540-72952-5_4)

# **Modeling and Specification**





# Foundation of a Framework to Support Compliance Checking in Construction Industry

Wuwei Shen<sup>1</sup>(✉), Guangyuan Li<sup>2,3</sup>, Chung-Ling Lin<sup>1</sup>,  
and Hongliang Liang<sup>4</sup>

<sup>1</sup> Department of Computer Science, Western Michigan University,  
Kalamazoo, MI, USA

{wuwei.shen, chung-ling.lin}@wmich.edu

<sup>2</sup> State Key Laboratory of Computer Science, Institute of Software,  
CAS, Beijing, China

ligy@ios.ac.cn

<sup>3</sup> School of Computer and Control Engineering, UCAS, Beijing, China

<sup>4</sup> Beijing University of Posts and Telecommunications, Beijing, China  
hliang@bupt.edu.cn

**Abstract.** Computer use is pervasive in our daily life and the increasing demand for computer applications has penetrated into various domains. Construction industry has become one of domains which are more reliable on the application of computer to implement regulatory compliance checking. Like many safety critical domains, the construction domain has its own set of international building codes on construction projects which must comply with. With the increasing complexity of construction projects, many manual compliance checking techniques have shown some serious issues. First, the manual techniques are error-prone due to human errors. Second, the complexity of a construction project exceeds the human limit to deal with. Third, the evolution of a construction project is inevitable and the human maintenance of a construction project is almost impossible because either the memory of the original project design has faded away or some development team members are gone. So, it has become a new trend to employ computers to support automatic regulatory compliance checking in construction industry. In this paper, we propose a novel framework to support compliance checking with the emphasis on the foundation of automatic regulatory compliance checking to certify whether a construction project complies with some international building codes. An example is illustrated how compliance checking is performed in the framework.

**Keywords:** Conformance checking · Compliance checking · Class diagram  
Instance diagram · International codes

## 1 Introduction

Computer use is pervasive in our daily life and the increasing demand for computer applications has penetrated into various domains. Construction industry has become one of the latest industries to be more reliable on the application of computer to

implement regulatory compliance checking. Like many safety critical domains, the construction domain has its own set of international codes on construction projects which must comply with. With the increasing complexity of construction projects, many manual compliance checking techniques have some serious issues [1]. First, the manual techniques are error-prone due to human errors. Second, the complexity of a construction project exceeds the human limit to deal with. Usually, a construction project should consider a set of international codes for different purposes. For instance, a construction project should comply with ICC 2009 [2] as part of the overall construction requirement and Energy code [3] for the energy conservation purpose. Third, the evolution of a construction project is inevitable and the human maintenance of a construction project is almost impossible because either the memory of the original project design has faded away or some development team members are gone. So, it has become a new trend to employ computers to support automatic regulatory compliance checking in construction Industry.

Compliance checking is not new in the computer science community since more and more safety critical industries require compliance checking [4–10, 40]. One of the most important reasons is that the failure of a project in these industries can have a serious consequence such as loss of life. Central to the compliance checking is that developers must demonstrate that a system or a project indeed complies with the relevant governmental or international standard documents. Thanks to advance in Model Driven Engineering (MDE) [11], the automatic compliance checking has become a feasible means in support of compliance checking in various safety critical domains [12–21].

Like compliance checking in the safety critical domains, the compliance checking in the construction industry has several obstacles which should be overcome. First, the ambiguity issue must be solved since most international codes in construction industry are written in a natural language like English [2]. Second, heterogeneity of the representation of construction projects from various construction companies when carrying out regulatory compliance checking is another touchy issue. Third, an appropriate integration methodology of a construction project into a set of international codes in construction industry should be sought.

In this paper, we propose to apply the conformance checking, which has been widely used in the Model Driven Architecture (MDA) [11], in support of automatic regulatory compliance checking in construction industry. Conformance checking aims to ensure that an instance model conforms to its original model. Due to the four levels of models, conformance checking can be performed at various levels. In the construction industry context, we will apply conformance checking to level zero and one. Namely, a domain model at level one is used to model a set of international codes and represented in a class diagram in the Unified Modeling Language (UML) [22]. An instance model denotes a specific construction project to be checked against international codes. In MDA, a domain model usually includes constraints in the Object Constraint Language (OCL) [23] since many kinds of constraints in a class diagram cannot be represented. The conformance checking in MDA ensures that an instance model is a valid instance of the domain model. In other words, an instance model should satisfy all constraints in OCL as well as constraints given in a class diagram such as the multiplicity restriction if the instance model conforms to a class diagram.

In this case, regulatory compliance checking of whether a project complies with a set of international codes is achieved.

Conformance checking at level one and zero is appropriate for conformance checking in construction industry due to the nature of international codes. Unlike some governmental and international standard documents, international codes in construction industry place more specific restrictions on a construction project. For instance, ICC 2009 requires “*Interior spaces intended for human occupancy shall be provided with active or passive heating systems capable of maintaining a minimum indoor temperature of 68° at a point 3 feet above the floor on the design heating day*” [2]. Obviously, this requirement can be well denoted by a class diagram, i.e. a domain model, where the constraint on an indoor temperature can be converted to an OCL constraint. Furthermore, a specific construction project can easily be converted to an instance model of a domain model, which models a set of international codes the project must comply with. As a result, conformance checking leverages the capability of compliance checking in construction industry by ensuring that a specific construction project satisfies all constraints in a class diagram. To lay out the foundation of a framework to support automatic regulatory compliance checking in construction industry, we formalize compliance checking by means of conformance checking in this paper.

The paper is organized as follows. Section 2 illustrates an example which demonstrates how conformance checking contributes to regulatory compliance checking in construction industry. We lay out the foundation of automatic regulatory compliance checking by formalization of conformance checking in Sect. 3. Section 4 presents some relevant work in support of compliance checking in various domains. We finally draw a conclusion and present some future work in Sect. 5.

## 2 Related Work

Compliance checking has been widely discussed in the construction community and manual checking of compliance checking has been proved to be time consuming, error prone, and expensive. Instead, automatic compliance checking has been proposed to tackle these problems and researchers in the construction community have presented various techniques in support of automatic compliance checking [24]. Tan et al. proposed an integration approach which combines building envelope design with building codes and simulation by means of decision tables [18]. Specifically, a building code is used to produce decision tables while information of a building project is represented as a tree-like structure via an Extended Building Information Model (EBIM) [25] including the building simulation output. Rules in the decision tables are checked for the design facts shown in EBIM to validate whether the building project complies with the building code. Ding et al. proposed an approach to represent building codes via object-based rules and design via an Industry Foundation Classes (IFC)-based internal model [26] to support compliance checking according to accessibility regulations. The Construction and Real Estate Network (CONENET) [27] project of Singapore employed a method to use semantic object in the FORNAX library to represent design information while properties and functions in FORNAX objects are used to denote regulatory rules [28]. The SMRTcodes project (International Code Council (ICC) 2012

[29]) of the ICC adopted an approach to represent ICC codes in a tuple format and represent designs using an IFC-based model for automatic compliance checking. An approach given by Zhang et al. concentrated on the integration of NLP and logic reasoning for automatic compliance checking [30]. Specifically, they employed NLP to generate a Prolog program to model a building code while converting design information for a construction project to a set of facts so running of the Prolog program can achieve the goal of compliance checking.

On the other hand, conformance checking is not new in the MDE community [31, 32]. The UML specification presents a metamodel for UML diagrams and the specification includes well-formedness rules to enforce the constraints on UML diagrams back to the first version [33]. Thus, conformance checking has been proposed to ensure that a UML model satisfies the UML metamodel according to the UML specification. Various tools have been implemented to support the syntax checking for a UML model by means of conformance checking which enforces not only multiplicity but also all OCL constraints in the UML metamodel [34].

With the rapid development of MDE, the UML specification allows to define a domain specific language via the UML profile mechanism. The UML profile mechanism facilitates conformance checking in support of a user defined metamodel which extends the UML metamodel [35]. In this way, developers are able to define a specific language via a UML profile and apply conformance checking to ensure that a specific model satisfies the UML project, i.e. the specific language, according to some purpose. For instance, OMG published a UML testing profile dedicated to Model-based testing. Conformance checking thus ensure whether a specific testing procedure follows the UML testing profile or not. Furthermore, conformance checking has been used to support forward engineering recently. As progress has been made in MDE in the past decade, many tools have the forward engineering feature which translate a design model into some executable skeletal code such as C program and Java program to reduce the implementation time. What programmers do is to fill out the detailed implementation in the skeletal code. Thus, a state captured during an execution time can be checked against its original class diagram via conformance checking. In this way, conformance checking ensures that each valid program state does not violate the class diagram given by the design phase [36].

### 3 An Illustrative Example of Compliance Checking

One challenging issue in support of regulatory compliance checking is how to denote the regulatory requirements given in a set of international codes. Due to the ambiguity issue in a document written in a natural language, various techniques have been proposed. In this paper, we employ a class diagram in UML to formalize the international codes in construction industry. However, before presenting the formalization of conformance checking, we would like to illustrate how a UML class diagram models the text in an international code such as ICC 2009.

To formalize the text in an international code, we need to read carefully the code's text to identify all concepts and their relationships. In order to systematically analyze the text, we first label an important noun or noun phrase as a concept and create a

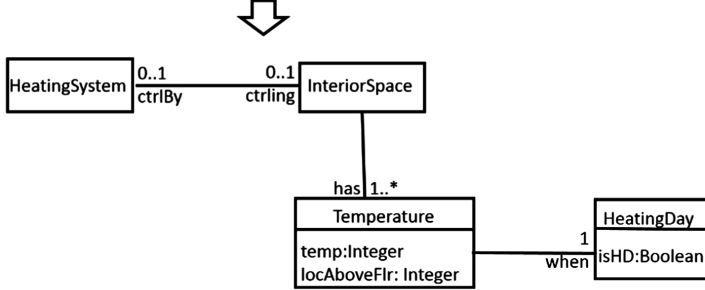
definition in a glossary if the noun or noun phrase is first encountered. We also study the relationship between these concepts to create an association relationship. In doing so, we are able to create a class diagram to model the text of an international code in construction domain.

As an example, we consider the text retrieved from Section 1204.1 in ICC 2009, which has the following text: “*Interior spaces intended for human occupancy shall be provided with active or passive heating systems capable of maintaining a minimum indoor temperature of 68° at a point 3 feet above the floor on the design heating day*”. In this text, we retrieve the following concepts: interior spaces, heating systems, design heating day, temperature, and above the floor. They are converted to UML classes *InteriorSpace*, *HeatingSystem*, *HeatingDay* and *Temperature* respectively. Specifically, we have attributes *temp* and *locAboveFlr* to denote the temperature at the specific point above the floor in class *Temperature*. Likewise, we have attribute *isHD* to denote whether a heating day is a design heating day or not in class *HeatingDay*. Also, we find an association between classes *InteriorSpace* and *HeatingSystem* to denote “*Interior spaces ... shall be provided with active or passive heating systems.*” The association between classes *InteriorSpace* and *Temperature* denotes the temperature at a specific location in an interior space required by the text “*...maintaining a minimum indoor temperature of 68° at a point 3 feet above the floor on the design heating day*”. Likewise, from the same part of the sentence, we have an association between classes *Temperature* and *HeatingDay* to model the temperature on a heating day. A complete class diagram is illustrated in Fig. 1(i) to illustrate the main concepts as well as their relationship extracted from the text from Section 1204.1.

However, some constraints represented in the text of an international code cannot be represented in a UML class diagram. In this case, we need to employ OCL to denote such constraints. For instance, in the 1204.1 section, we cannot represent the restriction on the minimum indoor temperature. Thus, we give an OCL constraint to enforce the restriction which is shown in Fig. 1(ii).

Once a class diagram is given based on the text of an international code, we are able to perform the automatic compliance checking by means of conformance checking in MDE to validate whether a construction project satisfies the code or not. In this way, we convert information from a construction project into an instance diagram and employ the conformance checking to achieve the compliance checking purpose. As an illustrative example, Fig. 2(i) shows an example of a construction project which complies with the text of Section 1204.1 while Fig. 2(ii) shows an example of another construction not complying with the same section. The former example shows two temperatures measured on two different design heating days and both temperatures satisfies the restriction of Section 1204.1. However, the latter example does not satisfy the OCL constraint since an instance of class *InteriorSpace*, i.e. object t2, has 60° as the value of the temperature at the point of 4 feet above the floor on a design heating day and this does not satisfy the OCL constraint; while the other temperature has the value of 76° at the point of 5 feet above the floor on the same design heating day.

Section 1204.1 in ICC 2009 has the following "Interior spaces intended for human occupancy shall be provided with active or passive heating systems capable of maintaining a minimum indoor temperature of 68 degree at a point 3 feet above the floor on the design heating day"



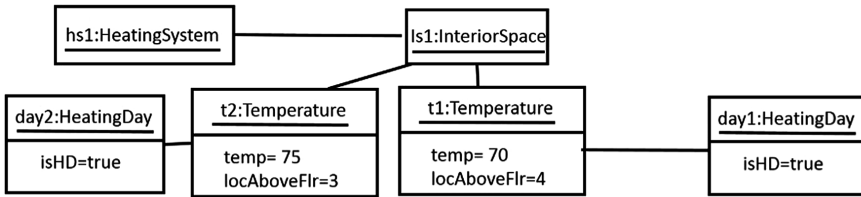
i) A class diagram/domain model for Section 1204.1

Contact InteriorSpace:

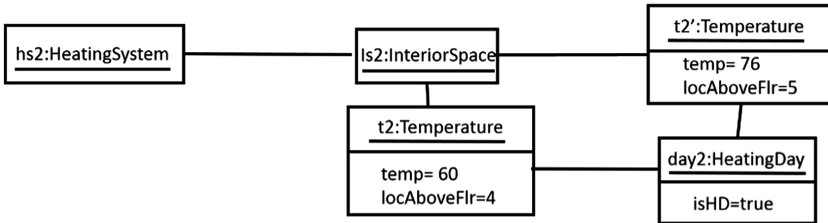
inv : self.ctrlBy.size()=1 implies self.has ->forall(e | e.when.isHD=true implies (e.temp>=68 and e.locAboveFlr>3))

ii) An OCL constraint attached to Class InteriorSpace

Fig. 1. A domain model and an OCL constraint for Section 1204.1



i) A valid instance model derived from a construction project



ii) An invalid instance model derived from a construction project

Fig. 2. Instance models showing two different construction projects

## 4 A Framework to Support Compliance Checking

According to the aforementioned example, we propose a new framework to support compliance checking in construction industry. The framework supports the formalization of a set of international building codes using UML class diagram as well as the representation of a construction project via an instance model. First, we employ a UML

class diagram to represent an international building code which a construction project should comply with. At the same time, we adopt the Object Constraint Language to denote the constraints in the code which cannot be represented in the class diagram. As mentioned above, in order to leverage the communication between the designers of a domain model and a construction project, a glossary table is provided to explain how the concepts from an international building code are derived. Second, a construction project is converted to an instance model according to a class diagram. Once these two inputs are provided, conformance checking is carried out and a compliance checking result is returned to designer of a construction project. The conformance checking part in the framework will be built on the UML2 APIs [37], OCL APIs [23], and Eclipse Modeling Framework (EMF) [38] in Eclipse. A diagram illustrating an overall structure as well as the flow of compliance checking of the framework is shown in Fig. 3.

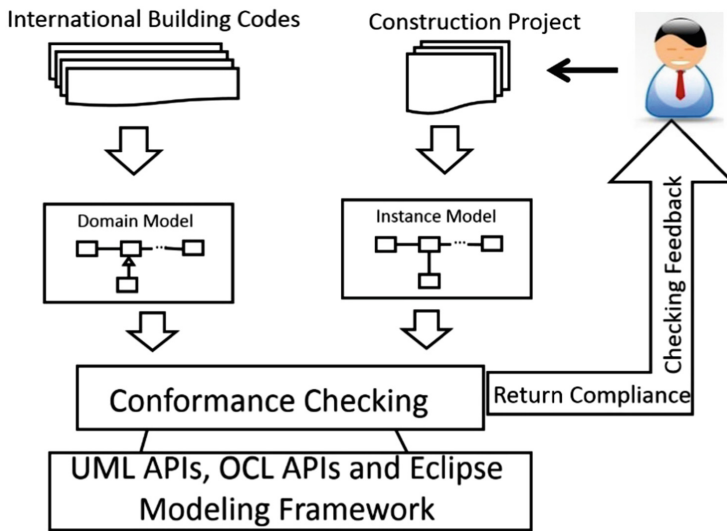


Fig. 3. A framework to support compliance checking

## 5 Formalization of Conformance Checking

In this section, we discuss the formalization of conformance checking for construction industry. As mentioned before, one integral part to conformance checking is a domain model which formally denotes a set of international codes and is given by a UML class diagram. Thus, we first formalize a UML class diagram as follows.

Let set  $\mathcal{A}$  be an alphabet and  $\mathcal{T}$  represents a set of type names. All string name set can be denoted as set  $\mathcal{S} \subseteq \mathcal{A}^+$ . Set  $\mathcal{T}$  includes all the types in UML such as *Integer* and all UML classes. Set  $Classes \subseteq \mathcal{S}$  denotes a set of class names. For each  $c \in Classes$ , we use  $t_c \in \mathcal{T}$  to denote the type which is the same as the class name  $c$ . The attributes of a class  $c \in Classes$  are defined as a set  $Attribute_c$  of signatures  $a: t_c \rightarrow t$  where  $a$  is the attribute and  $t_c$  is the type of the class (name)  $c$ . A set of associations is given by

- (i) a finite set of names  $Associations \subseteq \mathcal{S}$  and  
(ii) a function  $associates: \begin{cases} Associations \rightarrow Classes^+ \\ as \rightarrow \langle c_1, \dots, c_n \rangle \text{ with } n > 2 \end{cases}$

Let  $as \in Associations$  be an association with  $associates(as) = \langle c_1, \dots, c_n \rangle$ , which denotes an n-ary association  $as$  connects class  $c_1, \dots, c_n$ . Each association has a role name at a class end. Role names are defined by a function

$$roles: \begin{cases} Associations \rightarrow S^+ \\ as \rightarrow \langle r_1, \dots, r_n \rangle \text{ with } n > 2 \end{cases}$$

Intuitively,  $roles(as) = \langle r_1, \dots, r_n \rangle$  assigns each class  $c_i$  for  $1 \leq i \leq n$  participating in the association  $as$  a unique role name  $r_i$ . Let  $as \in Associations$  be an association with  $associates(as) = \langle c_1, \dots, c_n \rangle$ . The function  $multi(as) = \langle M_1, \dots, M_n \rangle$  assigns each class a non-empty  $M_i \subseteq \mathbb{N}$  with  $M_i \neq \{0\}$  for all  $1 \leq i \leq n$ , where  $\mathbb{N}$  denotes the set of all integers.

A generalization hierarchy  $\prec$  is a partial order on the set of classes  $Classes$ . For  $c_1, c_2 \in Classes$  we have  $c_1 \prec c_2$ , if and only if the class  $c_1$  is a child class of  $c_2$  and  $c_2$  is a parent class of  $c_1$  in UML. In order to collect all parents of a give class, we define the following function:

$$parent: \begin{cases} Classes \rightarrow 2^{Classes} \\ c \rightarrow \{c' | c' \in Classes \text{ and } c \prec c'\} \end{cases}$$

Then the full set of attributes of class  $c$  is given by set  $Attribute_c^*$  which contains all inherited attributes along a generation hierarchy as well as those directly defined in  $c$ .

$$Attribute_c^* = Attribute_c \cup \bigcup_{c' \in parent(c)} Attribute_{c'}$$

Another integral part to conformance checking is an instance model of a domain model. An instance model is used to model a specific construction project which consists of a set of objects as well as a set of links among them. Let  $I$  denote a mapping from the domain of all class diagrams, called the UML domain, to the domain of all instance diagrams, called the semantics domain. The semantics domain includes all values, such as *OclVoid*, types defined in OCL, such as *Integer*, and user defined classes. The domain of all instance diagrams includes some values such as  $\perp$ , an invalid value,  $\epsilon$ , a null value,  $\mathcal{Z}$  (all integer numbers), *true*, and *false* for OCL pre-defined values and types. So,  $I$  can be defined as  $I(Boolean) = \{true, false\} \cup \{\epsilon, \perp\}$ . Values  $\epsilon$  and  $\perp$  enable the evaluation of an expression which includes undefined and invalid values.

Furthermore, the semantics domain includes a set of instances of a class  $c \in Classes$  in a class diagram which is denoted by an infinite set  $oid(c) = \{o_1, o_2, \dots, o_n\}$ . Then the domain of a class  $c \in Classes$  is defined as  $I_{Classes}(c) = \bigcup \{oid(c') | c' \in Classes \text{ and } c' \prec c\}$ . Likewise, we can define  $I$  on the association. For each association  $as \in Associations$  with  $associates(as) = \langle c_1, \dots, c_n \rangle$ ,  $I_{Associates}(as) =$



$I_{Classes}(c_1) \times \dots \times I_{Classes}(c_n)$  denotes all possible links among objects instantiated from classes  $c_1, \dots, c_n$  according to association  $as$ . Next, an instance model for a class diagram  $D$  is a structure  $(D) = \{ \sigma_{Classes}, \sigma_{Attributes}, \sigma_{Associations} \}$  which consists of the following three parts:

- i. The finite set  $\sigma_{Classes}(c) \subseteq oid(c)$  represents all instance of class  $c$  in an instance model;
- ii. Function  $\sigma_{Attributes}(a)$  assigns attribute values to each object:  $\sigma_{Attributes}(a) : \sigma_{Classes}(c) \rightarrow I(t)$  for each  $a: t_c \rightarrow t \in Attribute_c^*$ ; and
- iii. The finite set  $\sigma_{Associations}(as) \subset I_{Associations}(as)$  denotes all valid links satisfying the multiplicity restriction defined for the association  $as$ .

A constraint  $ocl$  in OCL can be evaluated on an instance diagram  $ins$  derived from a class diagram  $d$  via the semantics function  $I: Constraints \times Environment \rightarrow \{true, false\}$ . An environment derived from an instance diagram and denoted as  $\pi$  consists of a state  $\sigma$  and a variable assignment  $\beta$ . The semantics function  $I$  is applied to the syntax of all OCL structures. For instance,  $I$  is applied to an if statement in OCL as follows and semantics for the rest OCL structure can be found [23]:

$$I(\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif}, \pi) = \begin{cases} I(e_2, \pi), & \text{if } I(e_1, \pi) = true; \\ I(e_3, \pi), & \text{if } I(e_1, \pi) = false; \\ \perp, & \text{otherwise} \end{cases}$$

For a specific constraint  $l$ , an environment  $\pi$  derived from an instance diagram  $ins$ , which is an instance model of a domain model  $d$ , and a mapping function  $I$ , if  $I(l, \pi) = true$ , then we say the environment  $\pi$  derived from instance diagram  $ins$  satisfies the domain model  $d$  based on the constraint  $l$ , denoted as  $ins \models_{(I, \pi)} l$ . If an instance diagram  $ins$  satisfies a domain model  $d$  based on all the constraints in  $d$ , which is  $\forall l \in d. Constraints$ , such that  $ins \models_{(I, \pi)} l$ , then we say the instance diagram conforms to the domain model denoted as  $ins \models_{(I, \pi)} d$ . The conformance relationship  $\models_{(I, \pi)}$  ensures that an instance diagram is a valid instance of the domain model. Assume a domain model  $d$  is derived from a code  $C$  and an instance diagram  $ins$  is based on a specific project denoted as  $S$ . If we have  $ins \models_{(I, \pi)} d$ , then we say the project  $S$  comply with the code  $C$ .

## 6 Conclusion and Future Work

In this paper, we give the theoretical foundation of compliance checking in construction industry as well as a case study illustrating how conformance checking in MDE aids to achieve the goal of compliance checking. One of the most important reasons we choose conformance checking is that several frameworks are available to carry out conformance checking in the MDE community. These frameworks have been proved efficient in the execution of conformance checking and we wish the efficiency of these frameworks can leverage the capability of compliance checking in construction industry and shorten design time for a construction project. The formal definition of

compliance checking provides the foundation of application of these frameworks in construction industry, ensuring that conformance checking carried out by the framework can be further proved to be correct in the support of compliance checking.

As a continuous step of the project, we will study the application of UML2 APIs, and OCL APIs in the Eclipse Modeling Framework (EMF) to support the conformance checking as illustrated earlier. EMF has been extensively applied as a framework to support MDA. Various tools and plugins such as IBM Rational Architect [39] and Eclipse Papyrus Plugin have been developed to support the EMF framework. The UML2 APIs are a set of APIs to process a UML model such as a class diagram while OCL APIs provide a set of APIs to aid OCL evaluation in a UML model. The UML2 and OCL APIs provide us with a programmatic method to automatically perform conformance checking and we will employ these APIs as well as the EMF framework to implement compliance checking in the next step.

**Acknowledgement.** This project is supported by the Georgeau Construction Research Institute at Western Michigan University. Li is supported by the National Natural Science Foundation of China (Nos. 61472406 and 61532019).

## References

1. Zhou, Z., Goh, Y.M., Shen, L.: Overview and analysis of ontology studies supporting development of the construction industry. *J. Comput. Civ. Eng.* **30**(6), 04016026 (2016)
2. International Code Council: International Codes (2009). <https://archive.org/stream/gov.law.icc.abc.2009#page/n177/mode/2up>. Accessed 23 July 2017
3. Model national energy code of Canada for houses 1997: Table A 3.3.1.1.—Prescriptive Requirements—Building Assemblies, 1st edn. National Research Council of Canada, Ottawa (1997)
4. Graydon, P., Habli, I., Hawkins, R., Kelly, T., Knight, J.: Arguing conformance. *IEEE Softw.* **29**, 50–57 (2012)
5. Kelly, T.: Arguing safety - a systematic approach to manage safety cases. Doctoral dissertation, Department of Computer Science, University of York (1998)
6. Woehrle, M., Lampka, K., Thiele, L.: Conformance testing for cyber-physical systems. *ACM Trans. Embed. Comput. Syst.* **11**(4), 1–23 (2013)
7. Hauge, A.A., Stølen, K.: A pattern-based method for safe control systems exemplified within nuclear power production. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 13–24. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33678-2\\_2](https://doi.org/10.1007/978-3-642-33678-2_2)
8. Pajic, M., Jiang, Z., Lee, I., Sokolsky, O., Mangharam, R.: Safety-critical medical device development using the UPP2SF model translation tool. *ACM Trans. Embed. Comput. Syst.* **13**(4), 1–26 (2014)
9. Rinehart, D.J., Knight, J.C., Rowanhill, J.: Current practices in constructing and evaluating assurance cases with applications to aviation. NASA/CR–2015-218678, January 2016
10. Denney, E., Pai, G.: A lightweight methodology for safety case assembly. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 1–12. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33678-2\\_1](https://doi.org/10.1007/978-3-642-33678-2_1)
11. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Proceedings of Future of Software Engineering (2007)

12. Dimyadi, J., Amor, R.: Automated building code compliance checking—where is it at. In: CIB WBC (2013)
13. Kasim, T.: BIM-based smart compliance checking to enhance environmental sustainability. Dissertation, Cardiff University (2015)
14. Lipman, R., Palmer, M., Palacios, S.: Assessment of conformance and interoperability testing methods used for construction industry product models. *Autom. Constr.* **20**(4), 418–428 (2011)
15. Nawari, N.O.: Automating codes conformance in structural domain. In: International Workshop on Computing in Civil Engineering (2011)
16. Solihin, W., Eastman, C.: Classification of rules for automated BIM rule checking development. *Autom. Constr.* **53**, 69–82 (2015)
17. Song, J.-K., Cho, G.-H., Ju, K.-B.: A study on the rule development for BIM-based automatic checking in a duct system. *Korean J. Air-Cond. Refrig. Eng.* **25**(13), 631–639 (2013)
18. Tan, X., Hammad, A., Fazio, P.: Automated code compliance checking for building envelope design. *J. Comput. Civ. Eng.* **24**(2), 203–211 (2010)
19. Yeoh, J.K., Wong, J.H., Peng, L.: Integrating crane information models in BIM for checking the compliance of lifting plan requirements. In: International Symposium on Automation and Robotics in Construction (2016)
20. Yurchyshyna, A., Zarli, A.: An ontology-based approach for formalisation and semantic organisation of conformance requirements in construction. *Autom. Constr.* **18**(8), 1084–1098 (2009)
21. Yurchyshyna, A., Zucker, C.F., Thanh, N.L., Lima, C., Zarli, A.: Towards an ontology-based approach for conformance checking modeling in construction. In: Proceedings of 24th W78 Conference-Bringing ITC Knowledge to Work (2007)
22. The Object Management Group (OMG): Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.4.1) (2011)
23. The Object Management Group: The Object Constraint Language (OCL) Version 2.4 (2014)
24. Pauwels, P., Van Deursen, D., Verstraeten, R., De Roo, J., De Meyer, R., Van de Walle, R., Van Campenhout, J.: A semantic rule checking environment for building performance checking. *Autom. Constr.* **20**(5), 506–518 (2011)
25. Eastman, C.M., Teicholz, P., Sacks, R.: BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors. Wiley, Hoboken (2011)
26. Building SMART: IFC Introduction. <http://buildingsmart.org/ifc/>
27. Tien, F.S., Zhong, Q.: Construction and Real Estate NETWORK (CORENET). *Facilities* **19** (11/12), 419–428 (2001)
28. Khemlani, L.: CORENET e-PlanCheck: Singapore’s automated code checking system. AECbytes “Building the Future” Article, 26 October 2005. [www.novacitynets.com/pdf/aecbytes\\_20052610.pdf](http://www.novacitynets.com/pdf/aecbytes_20052610.pdf). Accessed 23 July 2017
29. ICC (International Code Council): International Code Council AEC3, 26 October 2013. [http://www.aec3.com/en/5/5\\_013\\_ICC.htm](http://www.aec3.com/en/5/5_013_ICC.htm). Accessed 23 July 2017
30. Zhang, J., El-Gohary, N.: Automated information transformation for automated regulatory compliance checking in construction. *J. Comput. Civ. Eng.* **29**(4), B4015001 (2015)
31. Richters, M., Gogolla, M.: Validating UML models and OCL constraints. In: Proceedings of Unified Modeling Languages (UML 2000), York, England (2000)
32. Paige, R.F., Brooke, P.J., Ostroff, J.S.: Metamodel-based model conformance and multiview consistency checking. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **16**(3), 11 (2007)
33. The Object Management Group: Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.4.1) (2012)

34. Chavez, H.M., Shen, W., France, R.B., Mechling, B.A., Li, G.: An approach to checking consistency between UML class model and its Java implementation. *IEEE Trans. Softw. Eng.* **42**(4), 322–344 (2016)
35. Eclipse Foundation. <http://www.eclipse.org/modeling/mdt/>
36. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: *EMF: Eclipse Modeling Framework*. Pearson Education, London (2008)
37. Leroux, D., Nally, M., Hussey, K.: Rational software architect: a tool for domain-specific modeling. *IBM Syst. J.* **45**(3), 555–568 (2006)
38. Ayoub, A., Kim, B., Lee, I., Sokolsky, O.: A safety case pattern for model-based development approach. In: Goodloe, Alwyn E., Person, S. (eds.) *NFM 2012*. LNCS, vol. 7226, pp. 141–146. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-28891-3\\_14](https://doi.org/10.1007/978-3-642-28891-3_14)
39. Hunt, W.: Modeling, verification of cyber-physical systems. In: *National Workshop on High-Confidence Automotive Cyber-Physical Systems* (2008)
40. Denney, E., Pai, G.: Automating the assembly of aviation of safety cases. *IEEE Trans. Reliab.* **63**(4), 830–849 (2014)



# An Improved Reliability Testing Model Based on SOFL

Zhouxian Jiang<sup>1</sup>, Honghui Li<sup>2</sup>(✉), and Xuetao Tian<sup>1,2</sup>

<sup>1</sup> School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China

{zxjiang, hhli, xttian}@bjtu.edu.cn

<sup>2</sup> Engineering Research Center of Network Management Technology for High Speed Railway of MOE, Beijing, China

**Abstract.** With the rapid growing size of industry software, software reliability has become a necessary factor to measure software quality. Software testing, which can quantify and assess software reliability by establishing the model of software reliability, is an important method to verify software reliability. SOFL is an object-oriented structured formal language and engineering approach. SOFL consists of Condition Data Flow Diagram (CDFD) and formal description of CDFD called Module. It makes an accurate explanation of the software function, which is the basis of reliability testing. However, the reliability testing also need to consider the frequency of functional use, which cannot be expressed by SOFL. Therefore, this paper explores an improved method named DG-CDFD. It uses the state diagram to express the probability of transfer between different states based on CDFD, and builds a reliability testing model. Finally, this paper verifies the feasibility of this method through a case model building.

**Keywords:** Reliability testing model · SOFL · Formal specification  
Model building

## 1 Introduction

During the software development process, software testing is an indispensable link, which guides users to find errors in the program, verifies and ensures the reliability of software. The software reliability refers to: in the specified time, under the specified conditions, the ability of software that does not cause system failure. The probability measurement is called software reliability. Software reliability testing refers to testing the software in order to ensure and verify its reliability. It uses the software running profile, a statistical description of the actual use of software, to test the software randomly [1, 2]. The software reliability testing emphasizes the random selection of inputs according to the actual probability distribution, and emphasizes the coverage of the test requirements. In software reliability testing, test cases must be designed according to the probability distribution. So you can get more accurate reliability assessment, and find out the influence of fault in software reliability.

Formal methods can accurately define and standardize the requirements of the software. Formal methods have rigorous mathematical logic, solid theoretical basis, which are suitable for the reliability testing. SOFL (Structured Object-oriented Formal Language) is a formal language, also an engineering method, having the characteristics of high readability and modularity [3]. The general idea of using SOFL for software reliability testing is to use the SOFL to formalize the specification, then convert the specification into equivalent structural tables, generate test cases based on the menu, and test the software. However, the test process is mainly concerned about the realization of the function, but neglect the frequency of functional use, which derives from the innate defect of SOFL. Therefore, this paper explores an improved method of SOFL, which can reflect the probability and use it for the reliability test modeling. This method not only has the advantages of SOFL formalization, but also concisely expresses the different states in the reliability model. This method has a greater advantage for reliability testing. In this paper, we will study the process of constructing formalized testing model based on SOFL. First, the relevant knowledge of SOFL is briefly summarized, and then the improved method is described. Finally, an APP - PD software is taken as an example to verify the feasibility of the method.

## 2 The Method of Specification Formalization with SOFL

This section makes a brief introduction to the theory, and application of SOFL and so on, so that the readers can understand the method proposed in this paper.

### 2.1 The Composition of SOFL Formal Specification

On the theoretical basis of SOFL, it is considered that the formal method is equivalent to the process of designing a reliable software, that is, the formal method is the combination of formal specification analysis, refinement and formal verification. The role of SOFL is to construct a formal specification of requirements, as well as the design of software systems. The three elements of the SOFL requirements specification are the CDFD-Condition Data Flow Diagram, the Module, and the Process [3, 4]. CDFD describes the transition between each functional scene, as well as the flow of data, data reading and writing relationships. The module encapsulates each CDFD and expresses each data flow graph in a formalized language, and describes the pre-conditions and post-conditions. The process is every block of data flow diagram, including only the implementation of the process itself, with the input and output data stream, pre-conditions and post-conditions to form a complete CDFD. For example, Fig. 1 is an ATM CDFD [6]. Each CDFD corresponds to a Module. CDFD and Module constitute a complete formal specification.

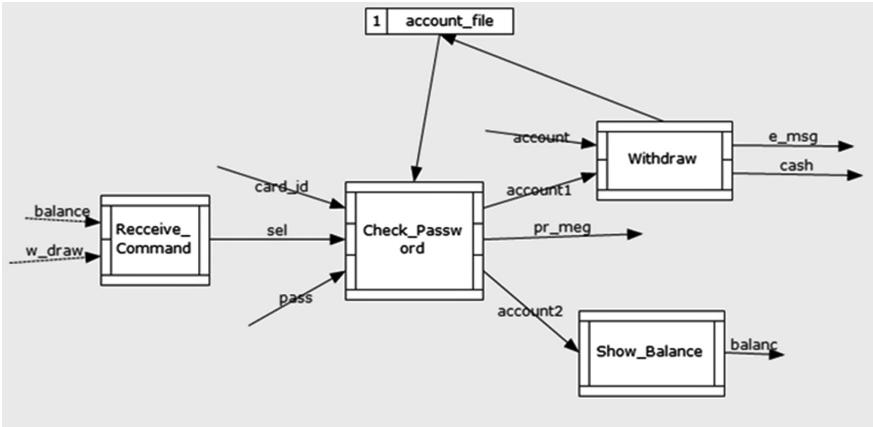


Fig. 1. An example of ATM machine CDFD.

### 2.2 The Composition of Reliability Testing Model

The software reliability testing model is an abstract description of software behavior and software architecture, as well as function use frequency. The software behavior includes the system input sequences, activities, conditions, output logic, data flow and so on. The software architecture includes component diagrams, and deployment diagrams. The testing model can be described by many ways. Figure 2 is an example of a testing model described by Markov Model. It reflects the probability of switching between different states (weather).

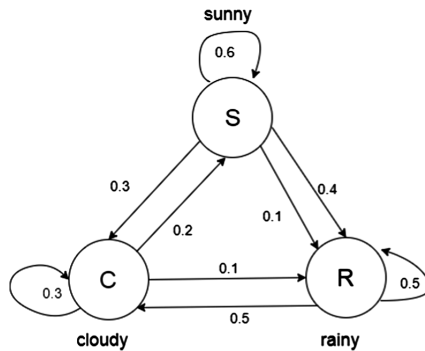


Fig. 2. An example of Markov model.

### 3 The Improved Method of Building a Test Model

After the SOFL formalizes the specification, the CDFD diagrams in the specification show different functional scenes and the links between the functional scenes. A CDFD diagram contains some processes. Each process can be considered as a state when we construct the model. Each process includes the following elements:

- (1) Input data streams, including control data streams and active data streams;
- (2) Output data streams, including control data streams and active data streams;
- (3) Data storage reading and writing;
- (4) Pre-conditions and post-conditions.

Therefore, the data stream of the input and output can be regarded as the conditions of the transition between one state and other state. When the input data stream satisfies the precondition, the state is activated. When the output data stream satisfies the post-condition and matches the next phase when the precondition of the neighbor state is transferred, it will be moved to the next state. The CDFD diagram can't reflect whether the input and output data streams can meet the pre-post condition. According to the references [6, 7] and the idea of the Markov chain model, we consider of using directed state diagram to improve the CDFD, This article gives a new definition called DG-CFDF (Directed graph of CDFD).

A DG-CDFD  $D$  refers to an ordered triple  $(V(D), E(D), P(D))$ , where:

- (1)  $V(D)$  is a set of nodes of the graph, which indicates the status of the software;
- (2)  $E(D)$  is a set of edges of the graph, which indicates the pre and post conditions of software state transfer;
- (3)  $P(D)(P \in [0,1])$  represents the probability of the software states transfer.

The probability of state transition can be based on expert experience, system log data and so on. In this paper, we use the average probability by the number of branches.

According to the definition, as shown in Fig. 1, an ATM machine CDFD can be transferred to a state diagram. The transformation of the state diagram is shown in Fig. 3.

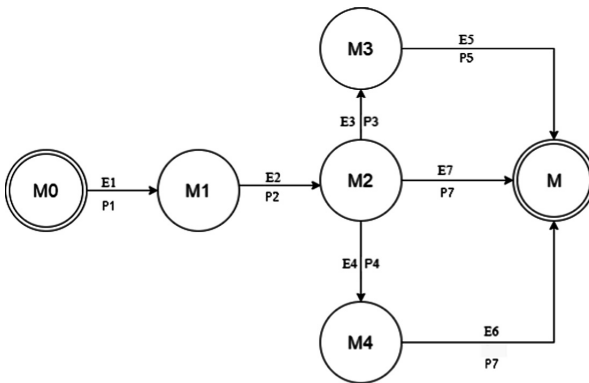


Fig. 3. Test model of ATM.



From Fig. 3 we can get the function of the scene of the three paths:

- (1)  $M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow M$ ;
- (2)  $M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow M$ ;
- (3)  $M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow M_4 \rightarrow M$ .

And it can be known that the probability of executing the path (1) is  $P_1 * P_2 * P_7$ , and the probability of the other paths are the same. The number of test cases generated for each path can be obtained. In addition, it should be noted that, because the CDFD diagram has a hierarchical characteristics, a CDFD diagram may be a layer of CDFD in a process of decomposition. Therefore, in the calculation of the current level CDFD, each path probability needs to consider the next level corresponds to the path of the probability of the process, and to be multiplied in order to get the correct probability.

After obtaining the complete test path of the system under test, the reliability test model based on the formal language is completed. A reliability test case can be generated by the model and be used in reliability testing.

## 4 Case Study of DG-CDFD

### 4.1 The Steps of Building Model

This paper mainly uses the formal verification method of formalization method, and uses SOFL to construct the software test model. The process is divided into three steps:

- (1) SOFL formal language is used to describe the specification of the measured system in detail to get a Completed functional scene.
- (2) The method of DG-CDFD is used to transform the formal specification into an equivalent model:
  - ① In SOFL CDFD, each process is defined as a state  $M_i$ , where  $i$  is the state identifier, and each CDFD has an initial state  $M_0$ , and a final state  $M$ .
  - ② In SOFL CDFD, the states can be connected by a directed edge  $E$ . Where  $E$  consists of the pre-condition and the post condition of the process in CDFD, and the direction of the transfer.
  - ③ In DG-CDFD, the probability between two states should be signed on the edge.
- (3) The model is used to get the software profile and its probability.

This section describes the game information inventory and shared software (referred to as “PD software”) as an example, and also describes the method and process of building the test model. In this section, we will briefly introduce the functional modules of the PD software and present the process of transforming the requirements of the PD system into the SOFL formal language model. In the process of formalizing

the specification, the SOFL supporting tool [3, 5], i.e. a tool of SOFL, is used. According to the functional scene of the system, the preconditions and post-conditions of each process are analyzed and explained. Finally, the functional scene of the PD system is transformed into a formalized reliability test model.

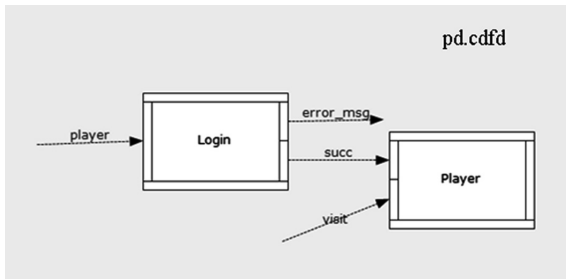
## 4.2 Formalize the Specification

We choose a mobile app PD as an example. PD system, is a community software, similar to the forum. There are three user identities: visitors, registered users and moderators. Visitors can access the system, browse news and topics and other contents, but can not post, make comments or like and take other operations.

When we formalized the specification of PD software, we adopted the top-down approach, that is, the functional scene of the topmost module of the whole software was transformed into formal description firstly, and then decomposed.

PD software to the top of the function is divided into two parts: (1) login function; (2) user access system functions.

If the user does not login the process, but directly access to the system, he will be in the state of “visitor”. If the user login the system, he will be in the “player” state. The player can apply for a moderator through the system function. So that you can completely cover the entire system of user identity. The function of the CDFD chart is shown in Fig. 4.



**Fig. 4.** The CDFD of PD system.

In Fig. 4, Login process describes the landing function. Player process describes the player to access the overall function of the system. The corresponding formalized module is shown in Table 1.

The decomposition process of the Login process gives you the next level of CDFD and its formal description. As the login is divided into three parts, one by phone number registration, login, the second is through the QQ login, the third is through the WeChat, the three process access to the data resource location is also different. After the login process is broken down, the created file is Login\_Decom.fModule, and the corresponding CDFD diagram is shown in Fig. 5.

**Table 1.** The module of PD System-pd.fModule.

---

```

pd.fModule
module pd; //Module name

var

type//data type

ext ;

process Login(player : int) error_msg: int| succ: int

pre player = 1;//precondition

post succ = -1 and error_msg = 1 or succ = 1 and error_msg = -1;//post condition

end_process;//process end

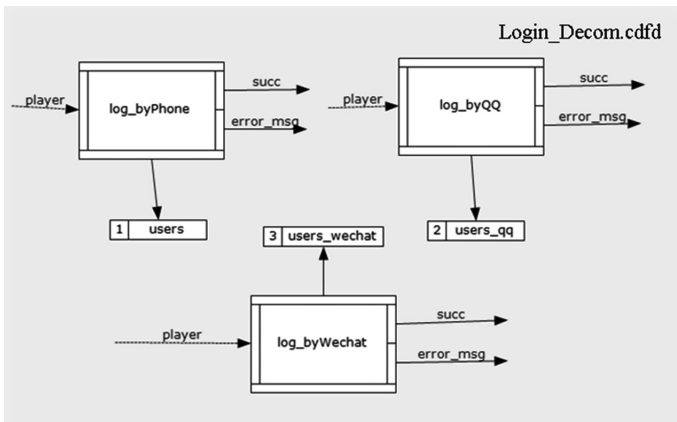
process Player(succ : int, visit : int)

pre succ = 1 or visit = 1; post;

end_process;

```

---

**Fig. 5.** The decomposed process CDFD of login.

Similarly, you can further mobile phone login, WeChat, QQ landing process to further layered refinement, and finally get a complete formal needs, due to limited space, where not repeat them one by one.

### 4.3 Modeling Process with the DG-CDFD

Using top-down modeling method, this paper transforms the top of CDFD graph into test model, and then transforms the next level CDFD one by one. This paper uses the login function module of the PD system for detailed description, and we take the average probability by the number of branches.

The top of the CDFD as shown in Fig. 4, be converted into a state graph as Fig. 6.

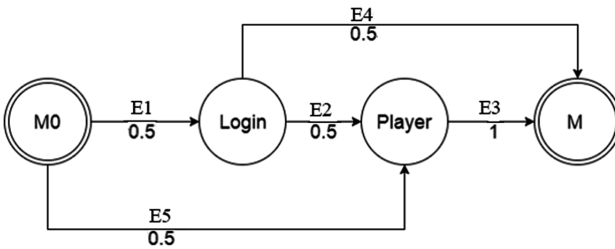


Fig. 6. Testing module of pd.cdfd.

According to Fig. 6 can be obtained in the process of the three functional paths:

- (1)  $M0 \rightarrow \text{Login} \rightarrow \text{Player} \rightarrow M$ , the path probability is  $0.5 * 0.5 * 1 = 0.25$ .
- (2)  $M0 \rightarrow \text{Player} \rightarrow M$ , the path probability is  $0.5 * 1 = 0.5$ .
- (3)  $M0 \rightarrow \text{Login} \rightarrow M$ , the path probability is  $0.5 * 0.5 = 0.25$ .

The decomposition of the known Login process is shown in Fig. 5, which is further transformed into a model using the model, as shown in Fig. 7.

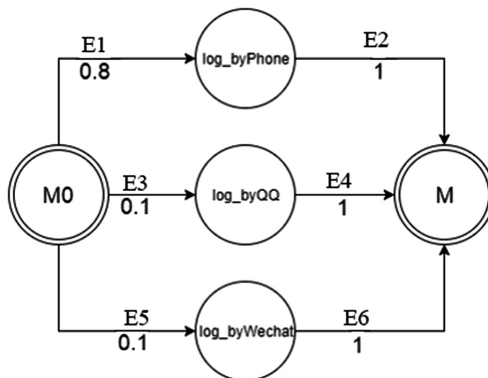


Fig. 7. Testing module of Login\_Decom.cdfd.

Then, we can get the whole model of the system by combining all the state diagrams, in which the login process is more detailed, as shown in Fig. 8.

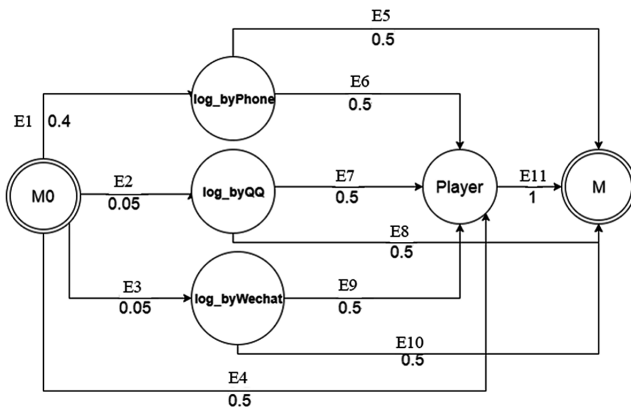


Fig. 8. PD system test model.

Finally, we can get the functional paths of the frequency information with the function:

- (1)  $M0 \rightarrow \text{log\_byPhone} \rightarrow \text{Player} \rightarrow M, P(1) = 0.2;$
- (2)  $M0 \rightarrow \text{log\_byQQ} \rightarrow \text{Player} \rightarrow M, P(2) = 0.005;$
- (3)  $M0 \rightarrow \text{log\_byWechat} \rightarrow \text{Player} \rightarrow M, P(3) = 0.005;$
- (4)  $M0 \rightarrow \text{Player} \rightarrow M, P(4) = 0.5;$
- (5)  $M0 \rightarrow \text{log\_byPhone} \rightarrow M, P(5) = 0.2;$
- (6)  $M0 \rightarrow \text{log\_byQQ} \rightarrow M, P(6) = 0.005;$
- (7)  $M0 \rightarrow \text{log\_byWechat} \rightarrow M, P(7) = 0.005.$

According to the functional path and its probability, we can design a reasonable test case. Assuming we need to design 1000 test cases, use case design can be shown in Table 2.

Table 2. Test cases table.

Serial number	Case name	Case number
1	Users use the phone number to log in successfully	200
2	Users use the QQ to log in successfully	50
3	Users use the Wechat to log in successfully	50
4	Visitors visit	500
5	The users failed to log in with the phone	200
6	The users failed to log in with the QQ	50
7	The users failed to log in with the Wechat	50

In summary, we can see that we only need to top-down hierarchically transform the CDFD graphs into state graphs with path probability, and then the probability of executing all paths can be calculated, meaning of the probability of executing each function scene.

## 5 Conclusion

Reliability testing technology has always been an important aspect of research in software testing field. Formal language provides a possible approach to this technology. SOFL language is a formal language for specification, providing the bases of the study of this paper. This paper explores a DG-CDFD method of using both state graph and SOFL formal specification to build a reliability testing model. so that SOFL can be better used in reliability testing. This paper based on the case, constructs a reliability test model of a mobile App, and verifies the feasibility of the method by comparison experiment. Formal engineering methods can make the development process more standard, the test process more accurate, and maintenance is more convenient. So the formal method applied to the reliability test is inevitable. However, there are still some limitations to this approach. Such as the extent of the application to a larger system, and the complexity of using it. In addition, it still needs further study that how to use SOFL testing model to create test cases automatically.

**Acknowledgements.** This work is supported by the National Key Scientific Instrument and Equipment Development Projects, China (No. 2013YQ330667).

## References

1. Gao, Z.J., Sun, F.C., Lu, L.: Software reliability test methods based on component composition and Markov process. *Adv. Mater. Res.* **660**, 169–173 (2013)
2. Chen, C.X., Li, M.A.: Research of software reliability test technology. *Comput. Eng. Des.* **31** (21), 4628–4631 (2010)
3. Liu, S.: *Formal Engineering for Industrial Software Development*, pp. 269–301. Springer, Berlin (2004). <https://doi.org/10.1007/978-3-662-07287-5>
4. Mat, A.R., Masli, A.B.: SOFL-based approach for requirements analysis of brain tumor treatment system. In: *International Conference on Computer & Information Science*. IEEE, pp. 885–888 (2012)
5. Li, M., Liu, S.: Integrating animation-based inspection into formal design specification construction for reliable software systems. *IEEE Trans. Reliab.* **65**(1), 88–106 (2016)
6. Li, L., Hou, S., Dai, Y., et al.: Automatic test case generation method based on DSG model. *J. Chin. Comput. Syst.* **36**(11), 2510–2514 (2015)
7. Huang, C.-Y.: Performance analysis of software reliability growth models with testing-effort and change-point. *J. Syst. Softw.* **76**(2), 181–194 (2005)
8. Li, H., Zhao, A., Zhang, D., et al.: Research on building software usage model based on UML model. *Int. J. Syst. Assur. Eng. Manag.* **10**, 1–9 (2017)



# A Framework Based on MSVL for Verifying Probabilistic Properties in Social Networks

Xiaobing Wang<sup>1</sup>, Liyuan Ren<sup>1</sup>, Liang Zhao<sup>1(✉)</sup>, and Xinfeng Shu<sup>2</sup>

<sup>1</sup> Institute of Computing Theory and Technology and ISN Laboratory,  
Xidian University, Xi'an 710071, People's Republic of China  
xbwang@mail.xidian.edu.cn, liyuan\_ren@foxmail.com, lzhao@xidian.edu.cn

<sup>2</sup> School of Computer Science and Technology, Xi'an University of Posts  
and Communications, Xi'an 710061, People's Republic of China  
shuxf@xupt.edu.cn

**Abstract.** In social networks, there are many phenomena related to randomness, such as interaction behaviors of users and dynamic changes of network structure. In this work, a framework based on MSVL (Modeling, Simulation and Verification Language) for verifying probabilistic properties in social networks is proposed. First, a hidden Markov model (HMM) is trained with the real social network dataset and implemented by MSVL. Then, an observed sequence is input into the trained HMM to obtain relevant information of the network, according to specialized algorithms. Next, a probabilistic property is specified with a formula of Propositional Projection Temporal Logic (PPTL). Finally, it is verified whether the MSVL model satisfies the PPTL property by model checking. An example of Sina Weibo is provided to illustrate how the framework works.

**Keywords:** MSVL · HMM · Social networks  
Probabilistic properties · Verification

## 1 Introduction

In recent years, social network has become a hot research topic in various fields, especially in the field of computer science. Studies on user behaviors in online social networks can effectively reveal the evolution of network structures and information transmission rules. It has theoretical significance to promote further development of social networks.

In a social network, user behaviors have strong randomness and thus various data have a stochastic characteristic. As for the research on randomness in social

---

This research is supported by the NSFC Grant Nos. 61672403, 61272118, 61402347, 61420106004, the Fundamental Research Funds for the Central Universities No. JBG160306, and the Industrial Research Project of Shaanxi Province No. 2017GY-076.

networks, some scholars study the issue based on statistical methods [1]. Other scholars study the issue based on formal techniques, such as B [2] and Petri net [3]. In our previous work, we study the privacy policy of social networks based on MSVL (Modeling, Simulation and Verification Language) [4]. However, none of these formal techniques has considered the randomness in social networks yet.

This paper utilizes MSVL to verify probabilistic properties in social networks. The main contributions are as follows. (1) A framework based on MSVL for verifying probabilistic properties in social networks is proposed. (2) The hidden Markov model (HMM) is applied to MSVL and the HMM-related algorithms are implemented with MSVL. (3) A social network application of Sina Weibo is illustrated to show how this framework works. With these contributions, MSVL can be used to verify probabilistic properties in social networks.

The rest of the paper is organized as follows. The theoretical basis of HMM, MSVL and MSV Platform is briefly introduced in the next section. Section 3 gives the framework based on MSVL for verifying probabilistic properties in social networks. Then, Sect. 4 is devoted to a case study of Sina Weibo to illustrate the whole procedure. Finally, related work is discussed in Sect. 5 and conclusions are drawn in Sect. 6.

## 2 Preliminaries

### 2.1 Hidden Markov Model

A hidden Markov model (HMM) is a doubly stochastic process with an underlying stochastic process that is not observable itself and can only be observed through another set of stochastic processes which produce the sequence of observed symbols [5]. An HMM is characterized by the compact notation  $\lambda = (S, V, \pi, A, B)$ , and the related character description is shown as follows.

- $S = \{s_1, s_2, \dots, s_N\}$ : the set of hidden states, where  $N$  is the number of states in the model.
- $V = \{v_1, v_2, \dots, v_M\}$ : the set of observed states, where  $M$  is the number of distinct observed states.
- $\pi = \{\pi_i\}$ : the initial state probability distribution, where  $\pi_i = Pr(s_i \text{ at } t = 1)$  is the probability that the state is  $s_i$  at time 1.
- $A = \{a_{ij}\}$ : the state transition probability distribution, where  $a_{ij} = Pr(s_j \text{ at } t + 1 | s_i \text{ at } t)$  is the probability that the state is  $s_j$  at time  $t + 1$  provided the state is  $s_i$  at time  $t$ .
- $B = \{b_i(k)\}$ : the observed state probability distribution in given state, where  $b_i(k) = Pr(v_k \text{ at } t | s_i \text{ at } t)$  is the probability that the observed state is  $v_k$  provided the state is  $s_i$  at time  $t$ .

For convenience, we use a triple  $\lambda = (\pi, A, B)$  to indicate the complete parameter for the model. Let  $O = (o_1, o_2, \dots, o_T)$  be an observed sequence where  $o_t \in V$  is the observed state at time  $t$ , and let  $Q = (q_1, q_2, \dots, q_T)$  be a state sequence where  $q_t \in S$  is the state at time  $t$ . The three classic problems of HMM are evaluation, decoding and training [5].



1. The evaluation problem: given a model  $\lambda$  and an observed sequence  $O$ , how to efficiently compute the  $P(O|\lambda)$ , the probability of the observed sequence given the model? It can be solved efficiently by the Forward-Backward algorithm.
2. The decoding problem: given a model  $\lambda$  and an observed sequence  $O$ , how to choose a corresponding state sequence  $Q$  which is optimal in certain meaningful sense? It can be solved efficiently by the Viterbi algorithm.
3. The training problem: given an observed sequence  $O$ , how to adjust the model parameters  $\lambda$  to maximize  $P(O|\lambda)$ ? It can be solved efficiently by the Baum-Welch algorithm.

## 2.2 MSVL and MSV Platform

MSVL is a temporal-logic-based programming language which can be used for modeling, simulation and verification of systems. It contains rich temporal operators, data structures and statements [6]. The basic temporal operators include the projection operator *prj*, the frame operator *frame*, and the wait operator *await*. The data types of MSVL include integer, float number, character, string, array types, list types, pointer types and struct types. The main statements of the language are given as follows, the detailed meaning of which statements is given in [6].

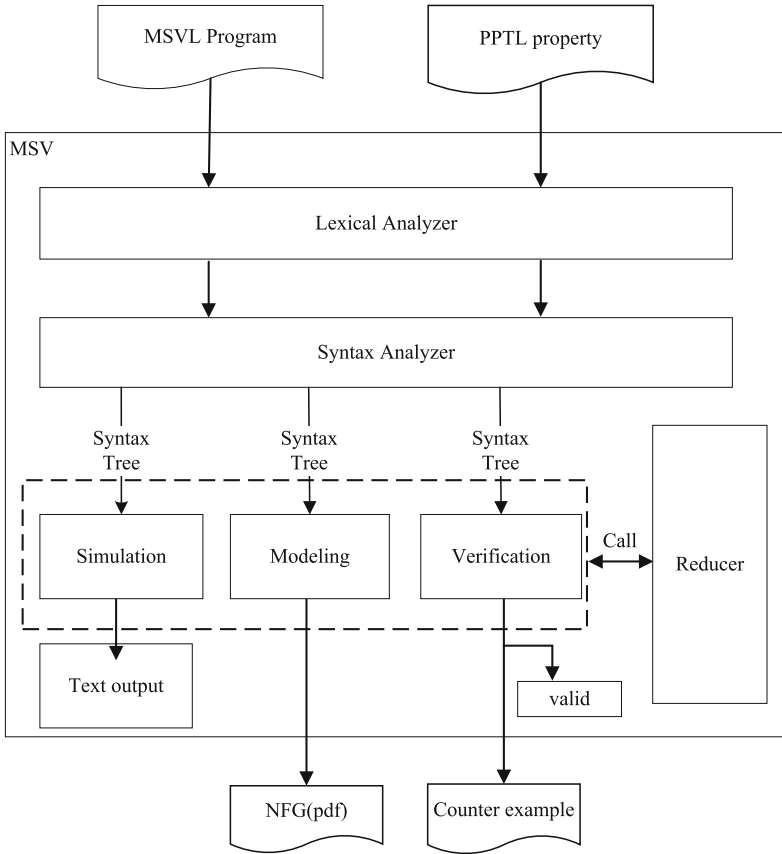
- |  |  |
|--|--|
| (1)Empty : <i>empty</i>                    | (2)Assignment : $x = e, x \leftarrow e$  |
| (3)Next : $\bigcirc p$                     | (4)Always : $\square p$                  |
| (5)Sometimes : $\diamond p$                | (6)Projection : $(p_1, \dots, p_m)prj p$ |
| (7)Sequence : $p ; q$                      | (8)Parallel : $p \parallel q$            |
| (9)Conditional : <i>if b then p else q</i> | (10)While : <i>while b do p</i>          |
| (11)StateFrame : <i>lbf(x)</i>             | (12)IntervalFrame : <i>frame(x)</i>      |
| (13)Await : <i>await(b)</i>                | (14)Finally : <i>fin(p)</i>              |

Propositional Projection Temporal Logic (PPTL) [7] is a proposition subset of Projection Temporal Logic (PTL), which excludes variables, quantifiers and predicates in PTL. The syntax of PPTL formulas  $P$  over a countable set  $Prop$  of atomic propositions is given as follows.

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

where  $p \in Prop$ ,  $P_1, \dots, P_m$  are well-formed PPTL formulas, and  $\bigcirc$  (next), *prj* (projection) are temporal operators. The detailed meaning of these formulas is given in [7]. For convenience, some derived formulas from elementary PPTL formulas are shown as follows. Abbreviations like *true* and  $\rightarrow$  are defined as usual.

$$\begin{array}{ll} \varepsilon \stackrel{\text{def}}{=} \neg \bigcirc true & more \stackrel{\text{def}}{=} \neg \varepsilon \\ P; Q \stackrel{\text{def}}{=} (P, Q)prj \varepsilon & \diamond P \stackrel{\text{def}}{=} true; P \\ \square P \stackrel{\text{def}}{=} \neg \diamond \neg P & fin(P) \stackrel{\text{def}}{=} \square(\varepsilon \rightarrow P) \end{array}$$



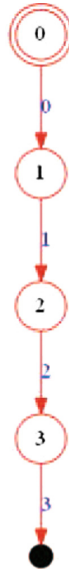
**Fig. 1.** The MSV platform

The language MSVL is the executable subset of PTL, so MSVL and PPTL are both in the PTL framework. This enables model checking within the same logic notations, i.e. the unified model checking approach [4]. In this approach, the system is modeled as a statement  $P$  using MSVL, and the desired property of system is specified by a formula  $\phi$  of PPTL. In order to check whether or not the model  $P$  satisfies the property  $\phi$ , the validity of the logic formula  $P \rightarrow \phi$  need to be checked. If  $P \rightarrow \phi$  is valid, the system satisfies the property, otherwise the system violates the property.

The corresponding tool for the unified model checking method is the MSV platform, developed by C++ under the Windows system. It uses Parser Generator as the lexical and syntax analysis module of MSVL program and Graphviz as the tool to diagram. As shown in Fig. 1, the MSV platform is the tool of executing MSVL programs, which includes three modes: simulation, modeling and verification. Simulation is to generate a model for the MSVL program, while modeling is to generate all models. Verification is based on the above unified



**Fig. 2.** The result of verification—satisfied



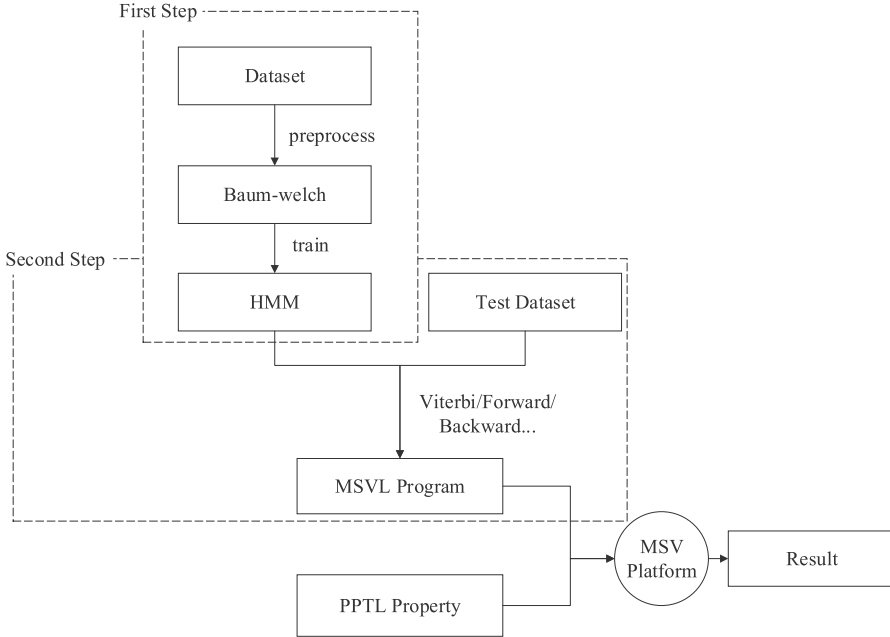
**Fig. 3.** The result of verification—unsatisfied

model checking approach. If the property is satisfied, the result is given that no dot or edge is marked red, as shown in Fig. 2. If not satisfied, a counterexample is given, as shown in Fig. 3.

### 3 Verifying Probabilistic Properties in Social Networks

In social networks, many phenomena happen randomly, such as user-interaction behaviors and structural changes of the network. This paper proposes a framework based on MSVL for verifying probabilistic properties in social networks, as shown in Fig. 4.

The specific process of this framework consists of four steps. First, a dataset related to randomness in a social network is crawled and preprocessed. Second, an HMM is trained with the dataset and implemented with MSVL. After that, on the basis of the trained HMM and an observed sequence, an appropriate HMM algorithm is implemented to obtain relevant information of the social network. Finally, probabilistic properties are specified with PPTL formulas and verified in the MSV platform.



**Fig. 4.** Method flow chart

Since an HMM is a probabilistic model using discrete variables to describe the states, the process of preprocessing the crawled dataset is to discretize the dataset. Then, the Baum-Welch algorithm and the dataset after preprocessing is used to train the HMM. In the HMM, if the number of the hidden states is  $N$  and the number of the observed state is  $M$ , the state set is  $S = \{s_1, s_2, \dots, s_N\}$  and the observed state set is  $V = \{v_1, v_2, \dots, v_M\}$ . The element  $\pi$  of the HMM is an array of length  $N$ ,  $A$  is a matrix of  $N * N$  and  $B$  is a matrix of  $N * M$ . So in the process of implementing the HMM with MSVL, the following data structures are used:

- an array *hidd\_list* to represent the hidden state set  $S$ ,
- an array *ob\_list* to represent the observed state set  $V$ ,
- an array *start\_p* to represent the initial state probability distribution  $\pi$ ,
- a two-dimensional array *trans\_p* to represent that the state transition probability distribution  $A$ , and
- a two-dimensional array *emit\_p* to represent the observed state probability distribution  $B$ .

Based on the trained HMM, an observed sequence and different algorithms, such as Viterbi, Forward and Backward, are used to obtain relevant information of social networks. Then, a probabilistic property is specified with a PPTL

formula and the MSVL program is executed in the MSV platform for verifying whether the property is satisfied or not. If the property is unsatisfied, a counterexample is given, as shown in Fig. 3.

## 4 Case Study

A better understanding of the notion of *user tie strength* can help social media sites serve their customers well, such as providing better recommendations and more effective friend management tools, which arises the problem of tie strength prediction [8]. In a social network, user interactive behavior is a random process. To certain extent, it reflects the tie strength between users. Therefore, the user interactive behavior can be viewed as the observed state and the user tie strength as the hidden state in an HMM. In this case study, we apply the proposed framework to a popular social network Sina Weibo to study the user tie strength and the user interactive behavior.

Our experiment collects the interaction behavior data of 5000 user pairs from April 2016 to April 2017 in Sina Weibo. The data contains three kinds of interaction behavior between users: like, comment and forward. In order to ensure the fairness of the data, five user pairs are selected randomly to train the model, respectively. The results of the training show that the five HMM models are very similar, and here we only show one trained model.

### 4.1 Modeling

In Sina Weibo, the relation between users can be observed such as following, followed and friend, but the tie strength between users cannot be obtained from these observations. Users can interact by liking, commenting and forwarding in Sina Weibo. Clearly, each interaction behavior can lead to different impacts on the tie strength of two users, and the frequency of the interaction behavior can also affect it. So we apply the framework to Sina Weibo, in which the tie strength between two users is taken as a Markov chain of the HMM and the interaction events between two users is taken as a random process of the HMM. In this way, the tie strength between two users can be obtained by analyzing the interactive events in a period of time, as shown in Fig. 5.

In order to establish the HMM, the data need to be represented through discrete values. Firstly, we define the tie strength between a user  $a$  and a user  $b$  as weak (0) or strong (1) [9]. So the set of hidden states  $S = \{0, 1\}$  and the number of states  $N$  is 2. Next, according to the interaction frequency, the interaction behaviors between users  $a$  and  $b$  are labeled by never (0), occasional (1), general (2) and regular (3). So the set of observed states  $V = \{0, 1, 2, 3\}$  and the number of distinct observed states  $M$  is 4. The specific method of labeling the user interaction behavior data is as follows.

In the dataset of Sina Weibo, the main interaction behaviors among users are *interbehavior* =  $\{Like, Comment, Forward\}$ . We take a week (7 days) as the time granularity, and use  $Like(a, b)$ ,  $Comment(a, b)$  and  $Forward(a, b)$  to

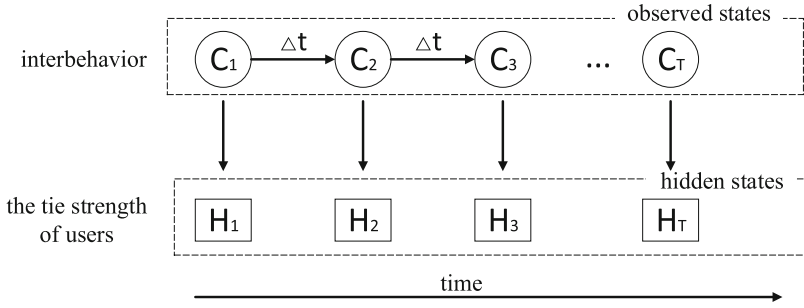


Fig. 5. The elements of the HMM

represent the number of like, comment and forward behaviors between two users  $a$  and  $b$  in a week, respectively. Then, the interaction data between the users  $a$  and  $b$  is quantified according to  $interaction(a, b) = \omega_1 * Like(a, b) + \omega_2 * Comment(a, b) + \omega_3 * Forward(a, b)$ . In order to facilitate the experiment, the value of  $\omega_i$  is weighted through subjective experience. Finally, the calculated result of  $interaction(a, b)$  is normalized and labeled by four grades.

According to the method described above, the data of user interaction behaviors between  $a$  and  $b$  labeled with a sequence of length 52, consisting of 0, 1, 2 and 3. Then, the data is trained by the Baum-Welch algorithm to obtain the HMM. The HMM is shown below.

$$\pi : \left(\frac{1}{2}, \frac{1}{2}\right)$$

$$A : \begin{pmatrix} 0.7703 & 0.2297 \\ 0.2156 & 0.7844 \end{pmatrix}$$

$$B : \begin{pmatrix} 0.5067 & 0.3876 & 0.1057 & 0.0034 \\ 0.0147 & 0.3418 & 0.3672 & 0.2763 \end{pmatrix}$$

Finally, the above HMM is implemented with MSVL. We make use of five arrays *hidd\_list*, *ob\_list*, *start\_p*, *trans\_p*, and *emit\_p* to represent the set of user tie strength, the set of user interactive behavior, the probability distribution of the initial user tie strength, the transition probability distribution of the user tie strength, and the observed state probability distribution in given state, respectively.

## 4.2 Verification

After building the HMM, we collect the interaction behavior data between two users  $a$  and  $b$  for 2 months from May to June, 2017. According to the model and the data, we intend to predict the changes of tie strength between  $a$  and  $b$  in the two months, and verify whether the probabilistic property of the tie strength meets the expectation. That is, given an HMM model and an observed sequence, to find the hidden state sequence that most likely produces such an observed sequence. This problem corresponds to the decoding problem of an

HMM, and the classic algorithm is Viterbi. So the above model and the user interaction data are used as the input of the MSVL program to implement the Viterbi algorithm. The data structures used are shown as follows.

- An array *ob\_seq* to represent the interactive behavior of users for two months,
- a two-dimensional array *result* to record the probability values, and
- a two-dimensional array *max* to record the state sequence that produces the maximum probability.

The specific calculation process is to calculate all possible paths and stores the probability in the *result* array to find out the path with the largest probability. That is, to find out the sequence of user tie strength change which is most likely to produce such observed sequence. In detail,  $result[x, 0]$  represents the probability of strong ties at the time of  $x + 1$ , and  $result[x, 1]$  represents the probability of weak ties at the time of  $x + 1$ . The specific MSVL program is shown in the Appendix.

In this case study, the following typical properties are described and verified.

**Property 1.** In the second moment, the probability of strong ties between two users  $a$  and  $b$  is greater than 0.7.

The property is described with a PPTL formula, where an atomic proposition needed is defined as:

$$p : result[1, 0] > 0.7.$$

The meaning of the proposition is that the probability of strong ties between  $a$  and  $b$  is greater than 0.7 in the second moment. The PPTL formula  $fin(p)$  need to be verified, and it is coded in MSV platform as  $fin p$ . It means that the program will check the satisfiability of propositional  $p$  in the last state of the interval. The verification result of the property is “unsatisfied” and a counterexample is given by the MSV platform as shown in Fig. 6. The above result shows that Property 1 is unsatisfied, i.e. the probability of strong ties between  $a$  and  $b$  is not more than 0.7 in the second moment.

**Property 2.** In the third moment, the probability of strong ties between user  $a$  and user  $b$  is less than weak ties between them.

The property is described with a PPTL formula, where an atomic proposition needed is defined as:

$$p : result[2, 0] < result[2, 1].$$

The PPTL formula  $fin(p)$  need to be verified, and it is coded in MSV platform as  $fin p$ . The verification result of the property is “satisfied” and the result is given by the MSV platform as shown in Fig. 2.

**Property 3.** The tie strength between user  $a$  and user  $b$  is non-decreasing.

The property is described with a PPTL formula, where a few atomic propositions needed are defined as:

$$\begin{aligned} p_0 : pre\_max\_node! &= -1; \\ p_1 : max\_node! &= -1; \\ q : pre\_max\_node &\leq max\_node. \end{aligned}$$



**Fig. 6.** The result of verification—unsatisfied

In the above atomic propositions, *max\_node* represents the tie strength between the users *a* and *b*, and *pre\_max\_node* represents the value of the previous state of the *max\_node*. The meaning of propositions  $p_0$  and  $p_1$  is that *pre\_max\_node* and *max\_node* are not initial values, respectively, and the proposition *q* means that the current tie strength of users is not less than the previous state. The PPTL formula  $\Box((p_0 \wedge p_1) \rightarrow q)$  need to be verified, and it is coded in MSV platform *alw((p0 and p1) => q)*. It means that when the tie strength between *a* and *b* is not initial, the tie strength always appears a non-decreasing trend. The verification result of the property is “satisfied” and the result is given by the MSV platform as shown in Fig. 2.

## 5 Related Work

In real-life systems, there are many phenomena that can be modeled by considering their stochastic characteristics. Scholars perform a lot of research in this field in recent years. In order to model the random phenomena, discrete and continuous time Markov chains (DTMCs and CTMCs), Markov decision processes (MDPs) and stochastic Petri nets are used in probabilistic model checking. Automated verification algorithms are proposed, which use the temporal logic CSL to specify performance and reliability measures for CTMCs [10]. Model checking algorithms are given for verifying DTMCs and CTMCs against specifications



written in probabilistic temporal logics PCTL and CSL, including quantitative properties with rewards [11]. A technique for automatically verifying quantitative properties of probabilistic systems is provided, which models both stochastic and nondeterministic behaviours with MDPs and specifies the probabilistic safety properties with PCTL and LTL [12]. Generalized stochastic Petri net, a modeling formalism that can be conveniently used for functional verification of complex models of discrete-event dynamic systems and for evaluation of their performance and reliability, is presented in [13].

Besides, temporal logics such as CTL and PTL are introduced with the probability to investigate the randomness phenomenon. The work [14] defines and explains model checking of probabilistic deterministic and nondeterministic systems using the probabilistic computation tree logics PCTL and PCTL\*. The method introduces a new probabilistic model checking procedure to check the compliance of target systems against some desirable properties, which specifies the properties with PCTL<sup>kc</sup> and reports the obtained verification results with a new version of interpreted systems. An approach to investigating probabilistic model checking based on PPTL is presented, which characterizes models with DTMCs and logic formulas of models with NFG<sub>inf</sub> [15]. And then the algorithm for determining and minimizing the nondeterministic NFG<sub>inf</sub> following the Safra's construction is given.

## 6 Conclusions

This paper presents an MSVL-based framework for verifying properties of probability in social networks. First, an HMM is trained according to a real social network dataset and implemented with MSVL. Next, an appropriate algorithm is selected, which uses the trained HMM and an observed sequence as the input, to obtain relevant information of social networks. Finally, a probabilistic property is specified by PPTL, so that whether the MSVL program satisfies the property can be verified. In future study, we will use other relevant algorithms of HMMs to obtain more information about the user tie strength based on the above case study, and verify probabilistic properties according to them. Besides, we plan to extend the framework with other phenomena related to randomness in social networks and attempt to introduce continuous HMMs into the framework.

## 7 Appendix: The MSVL Program

```

function main(){
  frame(N, M, start_p, trans_p, emit_p, ob_seq, cir_i, cir_j, result,
    max, max_node, pre_max_node) and (
    int pre_max_node<== -1 and int max_node<== -1 and skip;
    //N: the number of states
    //M: the number of distinct observed symbols
    int N and int M and skip;
    //Initialize N and M
    input(N) and skip;
    input(M) and skip;
    //cir_i,cir_j: control loop variables
    int cir_i <== 0 and skip;
    int cir_j <== 0 and skip;
    //start_p: the initial state probability distributions
    float start_p[N] and skip;
    //Initialize start_p
    while(cir_i < N){
      input(start_p[cir_i]) and skip;
      cir_i <== cir_i + 1 and skip
    };
    int cir_i <== 0 and skip;
    //trans_p: the state transition probability matrix
    float trans_p[N,N] and skip;
    //emit_p: the probability distribution matrix of observed values
    // in the given state
    float emit_p[N,M] and skip;
    //Initialize trans_p
    while(cir_i < N){
      while(cir_j < N){
        input(trans_p[cir_i,cir_j]) and skip;
        cir_j <== cir_j + 1 and skip
      };
      cir_j <== 0 and skip;
      cir_i <== cir_i + 1 and skip
    };
    cir_i <== 0 and skip;
    //Initialize emit_p
    while(cir_i < N){
      while(cir_j < M){
        input(emit_p[cir_i,cir_j]) and skip;
        cir_j <== cir_j + 1 and skip
      };
      cir_j <== 0 and skip;
      cir_i <== cir_i + 1 and skip
    };
};

```

```

//T: the length of the observed sequence
int T and skip;
//Initialize T
input(T) and skip;
//ob_seq: the observed sequence with length T
int ob_seq[T] and skip;
//Initialize ob_seq
cir_i <== 0 and skip;
while(cir_i < T){
    input(ob_seq[cir_i]) and skip;
    cir_i <== cir_i + 1 and skip
};
//Variables used for parameter passing
int *P <== start_p and skip;
int **A <== trans_p and skip;
int **B <== emit_p and skip;
int *O <== ob_seq and skip;
//Variables used for results
float result[T,N] and skip;
int max[T,N] and skip;
viterbi(N, M, P, A, B, T, O)
)
};
//Viterbi algorithm
function viterbi(int N, int M, int *start_p, int *A, int *B,
int T, int *O){
    frame(max, tmp, i, j, k, count, max_v) and (
        float tmp and skip;
        int i <== 0 and skip;
        int j <== 0 and skip;
        int k <== 1 and skip;
        int count <== 1 and skip;
        float max_v and skip;
        int max[T] and skip;
        //Step 1: initialization
        while (i < N){
            result[0, i] <== start_p[i] * B[i, 0[0]] and skip;
            i <== i + 1 and skip
        };
        //Step 2: recursion
        i <== 1 and skip;
        while (i < T){
            j <== 0 and skip;
            //Calculate the maximum probability
            while (j < N){

```

```

tmp <== result[i - 1, 0] * A[0, j] * B[j, 0[count]]
and skip;
max[i,j] <== 0 and skip;
k <== 1 and skip;
while(k < N){
  if(result[i - 1, k] * A[k, j] * B[j, 0[count]]>tmp)
  then{
    tmp <== result[i - 1, k] * A[k, j] * B[j, 0[count]]
    and skip;
    //Record the node with maximum probability
    max[i, j] <== k and skip
  }else{
    skip
  };
  //Record the maximum probability
  result[i, j] <== tmp and skip;
  output("result[" , i, "]"[" , j, "]: " , result[i, j], "\n")
  and skip;
  k <== k + 1 and skip
};
j <== j + 1 and skip
};
count <== count + 1 and skip;
i <== i + 1 and skip
};
//Step 3: termination
max_v <== result[T - 1,0] and skip;
max_node <== 0 and skip;
k <== 1 and skip;
//Find the maximum probability of the last state
while(k < N){
  if(result[T - 1,k] > max_v)
  then{
    max_v <== result[T - 1, k] and skip;
    pre_max_node <== k and skip;
    max_node <== k and skip
  }else{
    skip
  };
  k <== k + 1 and skip
};
//Step 4: path backtracking
k <== T - 1 and skip;
//Find the node with maximum probability
while(k >= 1){
  max[k] <== max_node and skip;
  pre_max_node<==max_node and skip;
  max_node <== max[k, max_node] and skip;
  k <== k - 1 and skip
};
pre_max_node<==max_node and skip;
//Output results
k <== 0 and skip;
while(k < T){
  output("max", k + 1, ":", max[k]) and skip;
  k <== k + 1 and skip
}
});
main()

```

## References

1. Zhang, J., Liu, B., Tang, J., et al.: Social influence locality for modeling retweeting behaviors. In: International Joint Conference on Artificial Intelligence, pp. 2761–2767. AAAI Press (2013)
2. Caviglione, L., Coccoli, M., Merlo, A.: Social Network Engineering for Secure Web Data and Services. IGI Global, Hershey (2013)
3. Ding, J., Cruz, I., Li, C.: A formal model for building a social network. In: IEEE International Conference on Service Operations, Logistics, and Informatics, pp. 237–242. IEEE Press, New York (2011)
4. Wang, X., Sun, T.: A method based on MSVL for verification of the social network privacy policy. In: Liu, S., Duan, Z. (eds.) SOFL+MSVL 2015. LNCS, vol. 9559, pp. 118–131. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-31220-0\\_9](https://doi.org/10.1007/978-3-319-31220-0_9)
5. Rabiner, L., Juang, B.: An introduction to hidden Markov models. *IEEE ASSP Mag.* **3**, 4–16 (1986)
6. Wang, X., Tian, C., Duan, Z., Zhao, L.: MSVL: a typed language for temporal logic programming. *Front. Comput. Sci.* **11**, 762–785 (2017)
7. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. *Acta Inform.* **45**, 43–78 (2008)
8. Tang, J., Chang, Y., Liu, H.: Mining social media with social theories: a survey. In: Applied Computing, Second Asian Applied Computing Conference, pp. 29–31. AACC, Nepal (2004)
9. Granovetter, M.: The strength of weak ties. *Am. J. Sociol.* **78**, 1360–1380 (1973)
10. Baier, C., Haverkort, B., Hermanns, H., et al.: Model-checking algorithms for continuous-time Markov chains. *IEEE Trans. Softw. Eng.* **29**, 524–541 (2003)
11. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-72522-0\\_6](https://doi.org/10.1007/978-3-540-72522-0_6)
12. Forejt, V., Kwiatkowska, M., Norman, G., et al.: Automated verification techniques for probabilistic systems. In: International School on Formal Methods for the Design of Computer, Communication and Software Systems, pp. 53–113. Advanced Lectures, Bertinoro (2011)
13. Balbo, G.: Introduction to generalized stochastic Petri nets. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 83–131. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-72522-0\\_3](https://doi.org/10.1007/978-3-540-72522-0_3)
14. Ciesinski, F., Größer, M.: On probabilistic computation tree logic. In: Baier, C., Haverkort, B.R., Hermanns, H., Katoen, J.-P., Siegle, M. (eds.) Validation of Stochastic Systems. LNCS, vol. 2925, pp. 147–188. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-24611-4\\_5](https://doi.org/10.1007/978-3-540-24611-4_5)
15. Yang, X.: Probabilistic model checking on propositional projection temporal logic. In: IMECS 2011. LNECS, vol. 2188, pp. 242–248 (2011)



# Implementing MapReduce with MSVL

Nan Zhang, Meng Wang, Zhenhua Duan<sup>(✉)</sup>, Cong Tian, and Jin Cui

Institute of Computing Theory and Technology, and ISN Laboratory,  
Xidian University, Xi'an 710071, China  
nanzhang@xidian.edu.cn, wangmengictt@163.com,  
{zhhdian, ctian}@mail.xidian.edu.cn

**Abstract.** This paper presents an approach to implementing MapReduce processes with Modeling Simulation and Verification Language (MSVL). This facilitates programmers not only to deal with large data sets but also to verify properties of programs in a convenient way.

**Keywords:** MapReduce · Multi-core · Parallel · Big data  
Cloud computing

## 1 Introduction

MapReduce [8] is a parallel computation model proposed by Google in Cloud Computing [4] to deal with large data sets. With this model, Map processes iterate over a large number of records ( $\langle \text{key}, \text{value} \rangle$  pairs) to extract something of interest from each record. Usually, intermediate results are firstly generated, and they are then shuffled and sorted. Further, Reduce processes aggregate these processed intermediate results, and produce final outputs. The key idea of MapReduce workflow is to provide a functional abstraction for Map and Reduce operations.

To process large data sets effectively, several supporting platforms such as Hadoop [1, 26], Storm [21], and Spark [22] supporting MapReduce are emerged. Hadoop is a distributed framework developed by Apache. The core of Hadoop is that a Hadoop Distributed File System called HDFS [5], a Hadoop distributed data base called HBase [17] and MapReduce technologies are implemented. Storm is a free and open source distributed realtime computation system [2]. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. It can be used with any programming language. Spark [3] is also an open source big data [14, 15] processing framework built around speed, ease of use, and sophisticated analysis. Spark has several advantages compared to other big data and MapReduce technologies like Hadoop and Storm.

MapReduce is a great solution for one-pass computations, but not very efficient for use cases that require multi-pass computations. Each step in the data processing workflow has one Map phase and one Reduce phase and we need to

---

The research is supported by National Natural Science Foundation of China under Grant Nos. 61420106004, 61732013, 61751207 and 61572386.

convert any use case into MapReduce pattern to leverage this solution. In most of cases, the intermediate processes are also called “partition” and “combine” processes. The output data of each step has to be stored in a distributed file system before the next step can begin. Hence, this approach tends to be slow due to replication and disk storage. Also, Hadoop solutions typically include clusters that are hard to set up and manage. It also requires the integration of several tools for different big data use cases. If you wanted to do something complicated, you would have to string together a series of MapReduce jobs and execute them in sequence. Each of those jobs was high-latency, and none could start until the previous job had finished completely. In contrast, Spark allows programmers to develop complex, multi-step data pipelines using Directed Acyclic Graph (DAG) pattern. It also supports in-memory data sharing across DAGs, so that different jobs can work with the same data.

Actually, Spark runs on top of existing Hadoop Distributed File System (HDFS) infrastructure to provide enhanced and additional functionality. It provides support for deploying Spark applications in an existing Hadoop v1 cluster or Hadoop v2 YARN cluster or even Apache Mesos [16]. We should view Spark as an alternative to Hadoop MapReduce rather than a replacement to Hadoop. It’s not intended to replace Hadoop but to provide a comprehensive and unified solution to manage different big data use cases and requirements.

MapReduce processes are usually realized by two functions `map()` and `reduce()` as well as two additional intermediate functions `partition()` and `combine()` which can in turn be implemented by any programming languages such as Java, C, C++, R [13] and Python etc.

In this paper, we would like to implement `map()`, `partition()`, `combine()` and `reduce()` functions in Modeling Simulation and Verification Language (MSVL) because of two reasons: (1) MSVL is a parallel programming language and can easily be used to deal with parallel computations. (2) MSVL programs are supported by a compiler MC [20] and a model checker at code level [18]. Hence, they can facilitate programmers to debug and verify their programs.

The main contribution of the paper is two-fold: (1) The principle for programming `map()` and `reduce()` as well as `partition()` and `combine()` with MSVL is summarized. (2) A unified model checking approach at code level [11, 18, 25] to verifying programs for MapReduce implementation is demonstrated.

The paper is organized as follows. Section 2 briefly presents MSVL. Section 3 concisely introduces MapReduce work flow. In Sect. 4, we summarize the principle for programming `map()`, `partition()`, `combine()` and `reduce()` with MSVL; then, a unified model checking approach at code level to verifying MSVL programs is demonstrated. In Sect. 5, as a case study, a sparse matrix multiplication [7] problem is given to show how MapReduce can be realized and verified with MSVL.

## 2 Preliminaries

### 2.1 Modeling, Simulation and Verification Language

Modeling, Simulation and Verification Language (MSVL) [9, 10] is an executable subset of Projection Temporal Logic (PTL) [10]. The following is a brief

introduction of syntax and semantics of MSVL. For more details, please refer to [10]. With MSVL, expressions can be treated as terms, and statements can be treated as formulas in PTL. The arithmetic and boolean expressions of MSVL can inductively be defined as follows:

$$e ::= n \mid x \mid \bigcirc e \mid \ominus e \mid f(e_1, \dots, e_n)$$

$$b ::= \text{true} \mid \text{false} \mid \neg b \mid b_0 \wedge b_1 \mid e_0 = e_1 \mid e_0 < e_1$$

where  $n \in \mathbb{R}$ , set of real numbers, and  $x \in \mathbb{V}$ , set of variables. The  $f()$  is a state function. The usual arithmetic operations such as  $+$ ,  $-$ ,  $*$  and  $\%$  can be viewed as two-arity functions. One may refer to the value of a variable at the previous state or the next one. The statements of MSVL can be inductively defined as follows.

	NAME	SYNTAX	SEMANTICS
1	Termination	empty	$\stackrel{\text{def}}{=} \varepsilon$
2	Assignment	$x := e$	$\stackrel{\text{def}}{=} \bigcirc x = e \wedge \text{len}(1)$
3	Positive immediate assignment	$x <== e$	$\stackrel{\text{def}}{=} x = e \wedge p_x$
4	State frame	$\text{lbf}(x)$	$\stackrel{\text{def}}{=} \neg \text{af}(x) \rightarrow \exists b : (\bigcirc x = b \wedge x = b)$
5	Interval frame	$\text{frame}(x)$	$\stackrel{\text{def}}{=} \square(\text{more} \rightarrow \bigcirc \text{lbf}(x))$
6	Next	next $\phi$	$\stackrel{\text{def}}{=} \bigcirc \phi$
7	Always	always $\phi$	$\stackrel{\text{def}}{=} \square \phi$
8	Conditional	if $b$ then $\phi_0$ else $\phi_1$	$\stackrel{\text{def}}{=} (b \rightarrow \phi_0) \wedge (\neg b \rightarrow \phi_1)$
9	Existential quantification	exist $x : \phi(x)$	$\stackrel{\text{def}}{=} \exists x : \phi(x)$
10	Sequential	$\phi_0 ; \phi_1$	$\stackrel{\text{def}}{=} \phi_0 ; \phi_1$
11	Conjunction	$\phi_0$ and $\phi_1$	$\stackrel{\text{def}}{=} \phi_0 \wedge \phi_1$
12	While	while $b \{ \phi \}$	$\stackrel{\text{def}}{=} (b \wedge \phi)^* \wedge \square(\varepsilon \rightarrow \neg b)$
13	Selection	$\phi_0$ or $\phi_1$	$\stackrel{\text{def}}{=} \phi_0 \vee \phi_1$
14	Parallel	$\phi_0 \parallel \phi_1$	$\stackrel{\text{def}}{=} \phi_0 \wedge (\phi_1 ; \text{true}) \vee (\phi_0 ; \text{true}) \wedge \phi_1 \vee \phi_0 \wedge \phi_1$
15	Projection	$(\phi_1, \dots, \phi_m) \text{ prj } \phi$	$\stackrel{\text{def}}{=} (\phi_1, \dots, \phi_m) \text{ prj } \phi$
16	Synchronous communication	await( $c$ )	$\stackrel{\text{def}}{=} \text{frame}(x_1, x_2, \dots, x_n) \wedge \square(\varepsilon \leftrightarrow c)$

MSVL supports structured programming and covers some basic control flow statements such as sequential, conditional and while-loop statements. Further, MSVL also supports non-determinism and concurrent programming by including selection, conjunction and parallel statements. Moreover, a framing technique is introduced to improve the efficiency of program executions and synchronize communication for parallel processes. In addition, MSVL has been extended in a variety of ways recently. For instance, multi-types including integer, float, string, char, pointer, array and struct, have been formalized and implemented [19]. Hence, typed variables and functions over the extended data domain can be defined. Further, a mechanism for internal and external function calls is formalized and implemented [24] recently. To execute MSVL programs, the normal form of MSVL programs has been defined in [10] and a compiler for MSVL has been developed [20]. Besides, to verify MSVL programs, a model checker at code level [18] has also been realized.



## 2.2 Cylinder Computation Model

In this section, Cylinder Computation Model (CCM) is briefly introduced. The details for CCM can be found in [23]. The sequence expression is defined as follows:

$$l ::= \emptyset \mid \epsilon \mid n \mid l_1 \cdot l_2 \mid l_1 \otimes l_2 \mid l^*$$

From the syntax, we can see that the sequence expression is an analogue of regular expressions where  $\emptyset$  denotes the empty set,  $\epsilon$  empty sequence expression and  $n \in N_0$ , set of natural numbers. The concatenation (“.”), sum (“ $\otimes$ ”) of any two sequence expressions, or Kleene closure (“\*”) of a sequence expression is also a sequence expression. The semantics of sequence expressions can be found in [23].

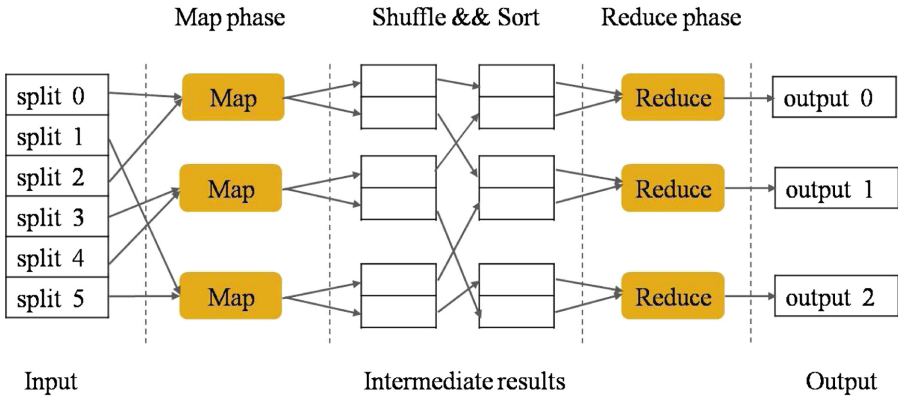
The syntax of CCM is given as follows:

$$CCM ::= \varphi \text{ ov } (l) \mid (CCM_1 \parallel CCM_2)$$

where  $\varphi$  is an MSVL program and  $l$  a sequence expression. Over (ov) and parallel ( $\parallel$ ) are temporal operators. For a CCM program  $\varphi \text{ ov } (l)$ , if it is satisfiable, the interpretation of  $\varphi$  is controlled by the sequence expression  $l$ . The semantics of Cylinder Computation Model is given in [23].

## 3 MapReduce Work Flow

MapReduce is a parallel mechanism for dealing with large scale data sets. Steps of a MapReduce execution are given in Fig. 1:



**Fig. 1.** Steps of MapReduce execution

where *split*  $i$  is an input data stream from external hardware file system while *output*  $j$  is an produced output data stream to external hardware storage. A group of Map operations first map the input data in the form  $\langle \text{in\_key}, \text{in\_value} \rangle$

in parallel into a list of an intermediate form  $\langle \text{inter\_key}, \text{inter\_value} \rangle$ . After that, in the Shuffle and Sort phases, a number of Partition and Combine operations are employed to generate another suitable intermediate form  $\langle \text{inter\_key}', \text{inter\_value}' \rangle$  so that, in the Reduce phase, a group of Reduce operations can eventually generate output results.

In summary, these signatures are as follows:

$\text{map}() : (\text{in\_key}, \text{in\_value}) \implies \text{list}(\text{inter\_key}, \text{inter\_value})$   
 $\text{partition}() : (\text{inter\_key}, \text{number of partitions}) \implies \text{partition for inter\_key}$   
 $\text{combine}() : (\text{inter\_key}, \text{inter\_value}) \implies \text{list}(\text{inter\_key}', \text{inter\_value}')$   
 $\text{reduce}() : (\text{inter\_key}', \text{list}(\text{inter\_value}')) \implies \text{list}(\text{out\_value})$

The principle of MapReduce is as follows:

- Programmers must specify:
  - $\text{map}(k, v) \implies \langle k', v' \rangle^*$
  - $\text{reduce}(k', v') \implies \langle k', v' \rangle^*$
 All values with the same key are reduced together.
- The following two operations are optionally:
  - $\text{partition}(k', \text{number of partitions}) \implies \text{partition for } k'$ 
    - Often a simple hash of the key, e.g.,  $\text{hash}(k') \bmod n$
    - Divides up key space for parallel reduce operations  $\text{combine}(k', v') \implies \langle k', v' \rangle^*$
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
- The execution framework handles everything else:
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts

Note that limited control over data and execution flow is permitted. All algorithms must be expressed in  $m$  (map),  $r$  (reduce),  $c$  (combine) and  $p$  (partition). We do not consider the following:

- Where mappers and reducers run;
- When a mapper or reducer begins or finishes;
- Which input a particular mapper is processing;
- Which intermediate key a particular reducer is processing;

In the above processes, the following points are useful: (1) Cleverly-constructed data structures are significant since they bring partial results together. (2) Sorting order of intermediate keys is useful because it facilitates us to control order in which reducers process keys. (3) Partitioner and combiner are sometimes necessary because we need to control a reducer to process a certain key. (4) Preserving state in mappers and reducers is necessary because we need to capture dependencies across multiple keys and values.

Notice that the following points are useful for local data aggregation. Ideal scaling characteristics requires the following: the data size is double, then the

running time is double while doubling the resources, then we can halve the running time. As a matter of fact, we cannot achieve the ideal scaling characteristics because synchronization requires communication which kills performance. Accordingly, we can avoid unnecessary communications by means of reducing intermediate data size via local aggregation and the help of combiners.

## 4 Principle of Programming with MSVL

We could use Hadoop or Sparke to build an environment for dealing with large data sets and make use of the functionality provided by them. However, in this paper, we concentrate on implementing MapReduce processes by means of MSVL programming. Hence, we only show some principles of programming `map()`, `partition()`, `combine()` and `reduce()` functions with MSVL.

### 4.1 Dealing with Hardware File Systems

Usually, large data sets are stored distributively on external storages. This subsection concerns with how to read from and write to external hardware file systems. Since we deal with large scale data sets, the input or output file could be very large. As a result, these files cannot be handled in memory locally and must be dealt in a workable way. Since MSVL can call C functions as external functions, we are able to call these functions on external file management such as `fopen()`, `fgets()`, `fputc()`, `fputs()` and `fclose()` etc. to handle input and output of batch data streams.

### 4.2 Partitioning Tasks into Parallel Computation Blocks

As you can see from Sect. 2, MSVL is a parallel programming language with several parallel or concurrent constructs such as  $P$  and  $Q$ ,  $P||Q$ ,  $(P_1, \dots, P_m) \text{ pr } j Q$  and  $P_1 \text{ ov } (l_1)|| \dots || P_m \text{ ov } (l_m)$  (CCM). Each of them has its own advantages to handle different parallel computation tasks. In particular, parallel operation ( $||$ ) can be used to implement mappers and reducers while the other ones are useful for realizing partitioner and combiners besides using other MSVL constructs. In addition, some shuffle and sort tasks may require various kinds of MSVL constructs. A framework of shuffle and sort relevant to mappers, reducers, partitioners and combiners is shown in Fig. 2.

### 4.3 Constructing Data Structures

MSVL is of plenty of data types used to build various data structures so that data can effectively be handled in local memory. In most of cases,  $\langle \text{key}, \text{value} \rangle^*$  can be managed by lists or arrays in MSVL. A complicated data consisting of several components can be constructed by the type `struct` in MSVL. In particular, the type pointer is useful to construct lists with an uncertain length.

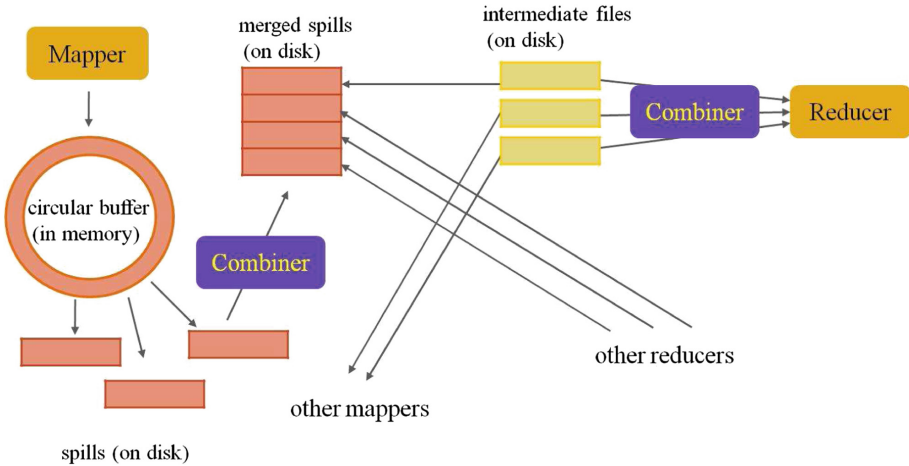


Fig. 2. Shuffle and sort

#### 4.4 Verifying Programs

Since a compiler and a model checker at code level are available in MSVL, these enable programmers to debug and verify their programs developed in MSVL from time to time. These characteristics make MSVL more useful than other programming languages such as Java, C/C++ and Python.

## 5 Case Study: Sparse Matrix Multiplication

Matrix multiplication, as a core problem in scientific computing, is challenging for large scale sparse matrices. This section presents a MapReduce framework for computing the product of two sparse matrices namely  $A \cdot B$ , which is implemented by MSVL programs. The matrix multiplication process includes the following three tasks [6]: (1) Represent the matrix as a list of nonzero entries. (2) Compute all products  $a_{i,k} \cdot b_{k,j}$  where  $a_{i,k}$  (resp.,  $b_{k,j}$ ) is the entry in matrix  $A$  (resp.,  $B$ ) in the  $i^{th}$  (resp.,  $k^{th}$ ) row and the  $k^{th}$  (resp.,  $j^{th}$ ) column. (3) Sum products for each entry  $i, j$ . The details of these tasks are given in the following subsections.

### 5.1 Matrix Representation

For the convenience of handling and storing, each nonzero entry in a matrix needs to be rewritten into a quadruple  $\langle row, col, value, matrixID \rangle$ . Suppose that two original sparse matrices  $A$  and  $B$  are stored in one file. In order to rewrite them, we define two *struct* **ReadData** and **Record** to store initial records and the processed data, respectively.

In the *struct* **ReadData**, the component **data** stores all original entries in the **rowNum**<sup>th</sup> row of the matrix **matrixID** and **idle** represents the state of this

memory block, where **idle = 1** indicates the memory block is idle, **idle = 2** means that entries of one row have been read into the idle memory block to be processed and **idle = 3** represents that the read entries in the memory block are being processed. In the *struct Record*, **data** stores all nonzero entries in one row which are in the form of the specified quadruple. Similar to **ReadData**, **idle = 1** means the memory block is idle, **idle = 2** represents that entries of one row are being processed and **idle = 3** indicates that the process of these entries has been completed.

```

1  struct ReadData
2  {
3      char *data and
4      int rowNum and
5      char matrixID and
6      int idle
7  };
8  struct Record
9  {
10     char *data and
11     int idle
12 };

```

We define a **ReadData** array  $rData[m]$  and a **Record** array  $wData[m]$  to store the records before and after data processing, respectively. With MSVL, the matrix rewriting process can briefly be specified as follows:

$$Init \stackrel{\text{def}}{=} ReadMatrix() || (HandleRecord(1) || \dots || HandleRecord(n)) || WriteRecord()$$

*ReadMatrix* reads entries of a row from the file into the memory when there is an idle memory block  $rData[i]$  ( $rData[i].idle = 1$ ). *HandleRecord* rewrites all nonzero entries stored in  $rData[i]$  ( $rData[i].idle = 2$ ) into the specified quadruple and stores them in an idle memory block  $wData[j]$  ( $wData[j].idle = 1$ ). After processing,  $rData[i].idle$  is reset to 1 and  $wData[j]$  to 3. Thus,  $n$  *HandleRecord* processes execute in parallel. Since there exists resource competition when these processes check whether there are records unprocessed in  $rData$  ( $rData[i].idle = 2$ ) and idle memory blocks in  $wData$  ( $wData[j].idle = 1$ ), the Bakery Algorithm [12] is adopted to solve the problem. *WriteRecord* writes the processed records in  $wData$  ( $wData[j].idle = 3$ ) into another file.

## 5.2 Map and Reduce of Matrix Multiplication in the First Phase

We use the column number of an entry in matrix  $A$  and the row number of an entry in matrix  $B$  as the key to map these entries into different groups. A process *Map1* is defined to write the nonzero entries with the same key in  $A$  into the same line and the ones in  $B$  into the next line in a file, which will be the input in the late reduce phase. We define a *struct KeyData* to store nonzero entries with the same key as follows:

```

1 struct KeyData
2 {
3     char *dataA and
4     char *dataB and
5     int key and
6     int idle
7 };
    
```

In the *struct* **KeyData**, the component **dataA** stores all nonzero entries in the **key**<sup>th</sup> column of matrix *A*, while **dataB** stores all nonzero entries in the **key**<sup>th</sup> row of matrix *B*. Further, **idle** has the same meaning with **idle** in *struct* **ReadData**. A **KeyData** array *rKeys*[*m*] is defined to store the records read from the file. The **reduce** process can be defined as follows:

$$Reduce1 \stackrel{\text{def}}{=} ReadRecord1() || (Reducer1(0) || \dots || Reducer1(n)) || WriteRecord()$$

Similar to *Init* defined in Subject. 5.1, *ReadRecords1* reads entries from the file into the memory line-by-line when there are idle memory blocks in *rKeys*. *Reducer1* computes all products of entries in *A* and *B* with the same key and stores them in *wData*.

### 5.3 Map and Reduce of Matrix Multiplication in the Second Phase

In this phase, the pair (*row, col*) is taken as the key to group all the products. After executing *Map2*, the entries with the same key are in the same line in a file. We define another **Record** array *rRec*[*m*] to store records read from the file. This **reduce** process can be defined as follows:

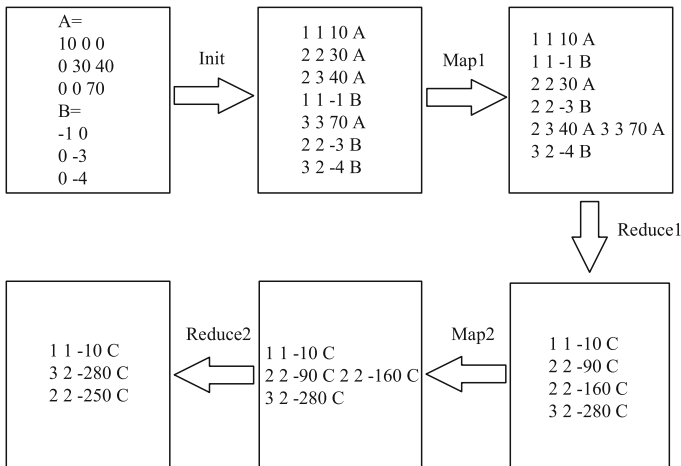


Fig. 3. Results of each phase

$$Reduce2 \stackrel{\text{def}}{=} ReadRecord2() || (Reducer2(0) || \dots || Reducer2(n)) || WriteRecord()$$

The function of *ReadRecords2* is similar with *ReadRecords1*. The difference between them is that *ReadRecords2* stores the entries in *rRec[m]*, while *ReadRecords1* stores them in *rKeys[m]*. *Reducer2* sums products with the same key to obtain final entries.

We compile our MSVL programs to binary executable codes with the MSVL compiler MC and obtain the results of each phase with two matrices *A* and *B* as its input as shown in Fig. 3.

All the MSVL programs solving sparse matrix multiplication problem in the MapReduce technique are placed in the Appendix A.

## 6 Conclusion

In this paper, we show that MSVL is useful for implementing MapReduce processes. In particular, parallel constructs in MSVL can be used to realize mappers and reducers. In the future, we will further deploy Hadoop, HDFS and HBase to provide a platform accompanying with MapReduce processes realized by MSVL to manipulate real distributed large scale data sets.

## Appendix A: MSVL Program of Sparse Matrix Multiplication

```

1  struct ReadData
2  {
3      char *data and
4      int row and
5      char matrixID and
6      int idle
7  };
8  function Max(int number[],int n, int RValue)
9  {
10     frame(i) and
11     (
12         int i<==0 and RValue:=0;
13         while(i<n)
14         {
15             if(RValue<number[i])then{RValue:=number[i]};
16             i:=i+1
17         }
18     )
19 };
20 function BakeryAlg(int tNum,int number[],int choosing[])
21 {
22     frame(j,temp) and
23     (
24         int j<==0 and int temp and skip;
25         choosing[tNum] := 1;

```

```

26     temp := 1 + extern Max(number, 5, RValue);
27     number[tNum]:=temp;
28     choosing[tNum] := 0;
29     while(j<5)
30     {
31         await(choosing[j]=0);
32         await(!(number[j]!=0 AND (number[j]<number[tNum] OR
33             (number[j]=number[tNum] AND j<tNum))));
34         j:=j+1
35     }
36 }
37 };
38 function StoreData(FILE *fp)
39 {
40     frame(i,brk) and
41     (
42         int i<==0, brk<==0 and skip;
43         while(i<5 AND brk=0){if(rData[i].idle=1)then{brk:=1}else{i:=i+1}};
44         if(rData[i].data=NULL)then
45         {rData[i].data:=(char*)malloc(bufferSize* sizeof(char))};
46         fgets(rData[i].data, bufferSize, fp) and skip;
47         if(rData[i].data[0]='B')then{rowA:=curRow-1};
48         if(rData[i].data[0]='A' OR rData[i].data[0]='B')then
49         {curRow:=1; matrixID:=rData[i].data[0]}
50         else {
51             if(rData[i].data[strlen(rData[i].data)-1]='\n')then
52             {rData[i].data[strlen(rData[i].data)-1]:='\0'};
53             rData[i].row:= curRow;
54             rData[i].matrixID:= matrixID;
55             rData[i].idle:=2;
56             curRow:=curRow+1
57         }
58     )
59 };
60 function ReadMatrix()
61 {
62     frame(fp) and
63     (
64         FILE *fp<==fopen("Matrix.txt", "r") and skip;
65         if(fp=NULL)then
66         {error:= 1 and output("Matrix.txt_cannot_be_opened.\n")}
67         else
68         {
69             while(error=0 AND feof(fp)=0)
70             {
71                 await(rData[0].idle=1 OR rData[1].idle=1 OR rData[2].idle=1
72                     OR rData[3].idle=1 OR rData[4].idle=1);
73                 StoreData(fp)
74             };
75             rowB:=curRow-1;
76             readOver:=1;
77             fclose(fp) and skip
78         }
79     )
80 };
81 struct Record
82 {
83     char *data and
84     int idle
85 };
86 function FormatCov(char* buf[],int rNum, int wNum)
87 {
88     frame(k,b) and
89     (
90         int k<==0 and char b[10] and skip;
91         if(wData[wNum].data=NULL)then
92         {wData[wNum].data:=(char*)malloc(bufferSize* sizeof(char))};
93         strcpy(wData[wNum].data, "") and skip;

```



```

94     while(buf[k]!=0)
95     {
96         if (strcmp(buf[k], "0") !=0) then
97         {
98             itoa(rData[rNum].row,b,10) and skip;
99             strcat(wData[wNum].data,b) and skip;
100            strcat(wData[wNum].data,"_") and skip;
101            itoa(k+1,b,10) and skip;
102            strcat(wData[wNum].data, b) and skip;
103            strcat(wData[wNum].data,"_") and skip;
104            strcat(wData[wNum].data, buf[k]) and skip;
105            strcat(wData[wNum].data,"_") and skip;
106            strcat(wData[wNum].data,&rData[rNum].matrixID) and skip;
107            strcat(wData[wNum].data,"\n") and skip
108        };
109        k:=k+1
110    }
111 }
112 };
113 function HandleLine(int tNum)
114 {
115     frame(i, j, brk, buf) and
116     (
117         BakeryAlg(tNum, rnumber, rchoosing);
118         await(rData[0].idle=2 OR rData[1].idle=2 OR rData[2].idle=2 OR
119             rData[3].idle=2 OR rData[4].idle=2 OR readOver=1);
120         int i<==0, j<==0, brk<==0 and char* buf[1000] and skip;
121         if(readOver=1 AND rData[0].idle!=2 AND rData[1].idle!=2 AND
122             rData[2].idle!=2 AND rData[3].idle!=2 AND rData[4].idle!=2)
123         then{ rnumber[tNum]:=0}
124         else{
125             while(i<5 AND brk=0)
126             {if(rData[i].idle=2)then{brk:=1}else{i:=i+1}};
127             rData[i].idle:=3;
128             rnumber[tNum]:= 0;
129             buf[j]:= strtok(rData[i].data, "_");
130             while(buf[j]!=0){j:=j+1; buf[j]:=strtok(NULL, "_");}
131             BakeryAlg(tNum, wnumber, wchoosing);
132             if(rData[i].matrixID='A' AND colA=0)then{colA:=j}
133             else{if(rData[i].matrixID='B' AND colB=0)then{colB:=j}};
134             await(wData[0].idle=1 OR wData[1].idle=1 OR wData[2].idle=1 OR
135                 wData[3].idle=1 OR wData[4].idle=1);
136             brk:=0; j:=0;
137             while(j<5 AND brk=0)
138             {if(wData[j].idle=1)then{brk:=1}else{j:=j+1}};
139             wData[j].idle:=2;
140             wnumber[tNum]:= 0;
141             FormatCov(buf, i, j);
142             rData[i].idle:=1; [j].idle:=3
143         }
144     )
145 };
146 };
147 function HandleMatrix(int tNum)
148 {
149     while(error=0 AND (rData[0].idle=2 OR rData[1].idle=2 OR
150         rData[2].idle=2 OR rData[3].idle=2 OR rData[4].idle=2 OR readOver=0))
151     {HandleLine(tNum)}
152 };
153 function WriteMatrix(char* fileName)
154 {
155     frame(fp, i, brk) and
156     (
157         FILE *fp<==fopen("MatrixInit.txt", "w") and int i, brk and skip;
158         if(fp=NULL)then
159         {error:=1 and output("MatrixInit.txt_cannot_be_opened.\n")}
160         else{
161             while(error=0 AND !(readOver=1 AND (rData[0].idle=1 AND

```

```

162         rData[1].idle=1 AND rData[2].idle=1 AND rData[3].idle=1 AND
163         rData[4].idle=1) AND (wData[0].idle=1 AND wData[1].idle=1 AND
164         wData[2].idle=1 AND wData[3].idle=1 AND wData[4].idle=1)))
165     {
166         i:=0;brk:=0;
167         await(wData[0].idle=3 OR wData[1].idle=3 OR wData[2].idle=3 OR
168             wData[3].idle=3 OR wData[4].idle=3);
169         while(i<5 AND brk=0)
170             {if(wData[i].idle=3)then(brk:=1)else{i:=i+1}};
171         fputs(wData[i].data,fp) and skip;
172         wData[i].idle:=1
173     };
174     fclose(fp) and skip
175 }
176 )
177 };
178 function RecordLen(int row,int col, int value, int RValue)
179 {
180     frame(b) and
181     (
182         char b[20] and skip;
183         itoa(row,b,10) and skip;
184         RValue:=strlen(b);
185         itoa(col,b,10) and skip;
186         RValue:=RValue+strlen(b);
187         itoa(value,b,10) and skip;
188         RValue:=RValue+strlen(b);
189         RValue:=RValue+5
190     )
191 };
192 function Count(FILE *fp)
193 {
194     frame(row,col,value) and (
195     int row, col, value and skip;
196     fscanf(fp,"%d_%d_%d_%c",&row,&col,&value,&matrixID) and skip;
197     while(!feof(fp)=0)
198     {
199         if(matrixID='A')then{
200             colDistrA[col-1]:=colDistrA[col-1]+RecordLen(row,col,value,RValue)}
201         else{
202             if(matrixID='B')then{ rowDistrB[row-1]:=
203                 rowDistrB[row-1]+RecordLen(row,col,value,RValue)}
204             };
205             fscanf(fp,"%d_%d_%d_%c",&row,&col,&value,&matrixID) and skip
206         }
207     }
208 };
209 function FindPos(FILE* fp,int rowOrCol, char matrixID)
210 {
211     frame(i,j) and
212     (
213         int i<=0,j<=0 and skip;
214         fseek(fp,0,0) and skip;
215         while(i<rowOrCol-1)
216         {
217             if(i<curKey-1)then
218                 {fseek(fp,colDistrA[i]+rowDistrB[i],1) and skip}
219             else{
220                 fseek(fp,wColA[i],1) and skip;
221                 j:=wColA[i];
222                 while(j<colDistrA[i]){fprintf(fp,"_") and j:=j+1};
223                 fseek(fp,wRowB[i],1) and skip;
224                 j:=wRowB[i];
225                 while(j<rowDistrB[i]){fprintf(fp,"_") and j:=j+1}
226             };
227             i:=i+1
228         };
229         fseek(fp,wColA[i],1) and skip;

```

```

230     if(matrixID='B')then
231     {
232         j:=wColA[i];
233         while(j<colDistrA[i]){fprintf(fp,"_") and j:=j+1};
234         fseek(fp,wRowB[i],1) and skip
235     };
236     if(curKey<rowOrCol)then{curKey:=rowOrCol}
237 }
238 };
239 function Write2File(FILE* fp,int row, int col, int value, int matrixID)
240 {
241     if(matrixID='A') then
242     {
243         FindPos(fp, col, 'A');
244         fprintf(fp,"%d_%d_%d_%c", row, col, value, matrixID) and skip;
245         wColA[col-1] := wColA[col-1]+RecordLen(row,col,value,RValue);
246         if(colDistrA[col-1]=wColA[col-1]+1)then
247         {wColA[col-1] := wColA[col-1]+1;fprintf(fp,"\n") and skip}
248         else{fprintf(fp,"_") and skip}
249     }
250     else
251     {
252         if(matrixID='B')then
253         {
254             FindPos(fp, row, 'B');
255             fprintf(fp,"%d_%d_%d_%c", row, col, value, matrixID) and skip;
256             wRowB[row-1] := wRowB[row-1]+RecordLen(row,col,value,RValue);
257             if(rowDistrB[row-1]=wRowB[row-1]+1)then
258             {wRowB[row-1] :=wRowB[row-1]+1;fprintf(fp,"\n") and skip}
259             else{fprintf(fp,"_") and skip}
260         }
261     }
262 };
263 function Map1()
264 {
265     frame(rFp,wFp,row,col,value) and
266     (
267         FILE *rFp and FILE *wFp and int row,col,value and skip;
268         rFp:=fopen("MatrixInit.txt","r");
269         if(rFp=NULL)then
270         {error:=1 and output("MatrixInit.txt_cannot_be_opened.\n")}
271         else
272         {
273             wFp:=fopen("map1.txt","w");
274             if(wFp=NULL)then
275             {error:=1 and output("map1.txt_cannot_be_opened.\n")}
276             else{
277                 Count(rFp); i:=0;
278                 while(i<colA)
279                 {
280                     if(colDistrA[i]!=0)then{colDistrA[i]:=colDistrA[i]+1};
281                     if(rowDistrB[i]!=0)then{rowDistrB[i]:=rowDistrB[i]+1};
282                     i:=i+1
283                 };
284                 fseek(rFp,0,0) and skip;
285                 fscanf(rFp,"%d_%d_%d_%c",&row,&col,&value,&matrixID) and skip;
286                 while(feof(rFp)=0)
287                 {
288                     Write2File(wFp,row,col,value,matrixID);
289                     fscanf(rFp,"%d_%d_%d_%c",&row,&col,&value,&matrixID) and skip
290                 };
291                 fclose(wFp) and skip
292             };
293             fclose(rFp) and skip
294         }
295     )
296 };
297 struct KeyData

```

```

298 {
299     char *dataA and
300     char *dataB and
301     int idle
302 };
303 function ReadRecord1()
304 {
305     frame(fp,i,j,brk) and
306     (
307         FILE *fp<==fopen("map1.txt","r") and skip;
308         int i<==0, j<==0, brk<==0 and skip;
309         if(fp=NULL)
310         then{error:=1 and output("map1.txt_cannot_be_opened.\n")}
311         else
312         {
313             while(error=0 AND feof(fp)=0)
314             {
315                 await(rKeys[0].idle=1 OR rKeys[1].idle=1 OR rKeys[2].idle=1
316                     OR rKeys[3].idle=1 OR rKeys[4].idle=1);
317                 i<==0 and j<==0 and brk<==0 and skip;
318                 while(i<5 AND brk=0)
319                 {if(rKeys[i].idle=1)then{brk:=1}else{i:=i+1}};
320                 if(rKeys[i].dataA=NULL)then
321                 {rKeys[i].dataA:=(char*)malloc(bufferSize* sizeof(char));}
322                 if(rKeys[i].dataB=NULL)then
323                 {rKeys[i].dataB:=(char*)malloc(bufferSize* sizeof(char));}
324                 fgets(rKeys[i].dataA, bufferSize, fp) and skip;
325                 fgets(rKeys[i].dataB, bufferSize, fp) and skip;
326                 if(rKeys[i].dataA[0]>'0' AND rKeys[i].dataA[0]<='9')
327                 then{rKeys[i].idle:=2}
328             };
329             readOver:=1;
330             fclose(fp) and skip
331         }
332     )
333 };
334 function MulRecord(int wNum, int valueA[],int valueB[],
335                   char *row[], char*col[])
336 {
337     frame(i,k,b) and
338     (
339         int i<==0,k<==0 and char b[10] and skip;
340         if(wData[wNum].data=NULL)then
341         {wData[wNum].data:=(char*)malloc(bufferSize*sizeof(char));}
342         strcpy(wData[wNum].data,"") and skip;
343         while(row[i]!=0)
344         {
345             k:=0;
346             while(col[k]!=0)
347             {
348                 strcat(wData[wNum].data,row[i]) and skip;
349                 strcat(wData[wNum].data,"_") and skip;
350                 strcat(wData[wNum].data,col[k]) and skip;
351                 strcat(wData[wNum].data,"_") and skip;
352                 itoa(valueA[i]*valueB[k],b,10) and skip;
353                 strcat(wData[wNum].data,b) and skip;
354                 strcat(wData[wNum].data,"_") and skip;
355                 strcat(wData[wNum].data,"C") and skip;
356                 strcat(wData[wNum].data,"\n") and skip;
357                 k:=k+1
358             };
359             i:=i+1
360         }
361     )
362 };
363 function RecordReducer1(int tNum)
364 {
365     frame(i,j,m,n,row,col,valueA,valueB,value1,value2,brk,tmp) and

```

```

366 (
367   BakeryAlg(tNum, rnumber, rchoosing);
368   await((rKeys[0].idle=2 OR rKeys[1].idle=2 OR rKeys[2].idle=2 OR
369         rKeys[3].idle=2 OR rKeys[4].idle=2) OR readOver=1);
370   int i<:=0, j<:=0, m<:=0, n<:=0, brk<:=0 and char*tmp and
371   int valueA[1000] and int valueB[1000] and char* row[1000] and
372   char* col[1000] and char* value1 and char* value2 and skip;
373   if(readOver=1 AND rKeys[0].idle!=2 AND rKeys[1].idle!=2 AND
374     rKeys[2].idle!=2 AND rKeys[3].idle!=2 AND rKeys[4].idle!=2)
375   then{rnumber[tNum]:=0}
376   else{
377     while(i<5 AND brk=0)
378     {if(rKeys[i].idle=2)then{brk:=1} else{i:=i+1}};
379     rKeys[i].idle:=3;
380     rnumber[tNum]:= 0;
381     if(rKeys[i].dataA=NULL)then
382     {rKeys[i].dataA:=(char*)malloc(bufferSize*sizeof(char))}
383     row[m]:=strtok(rKeys[i].dataA, "_");
384     while(row[m]!=0)
385     {
386       strtok(NULL, "_") and skip; value1:=strtok(NULL, "_");
387       valueA[m]:=atoi(value1); strtok(NULL, "_") and skip;
388       m:=m+1; row[m]:=strtok(NULL, "_")
389     };
390     tmp:=strtok(rKeys[i].dataB, "_");
391     while(tmp!=0)
392     {
393       col[n]:=strtok(NULL, "_"); value2:=strtok(NULL, "_");
394       valueB[n]:= atoi(value2); strtok(NULL, "_") and skip;
395       n:=n+1; tmp:=strtok(NULL, "_")
396     };
397     col[n]:=0;
398     BakeryAlg(tNum, wnumber, wchoosing);
399     await(wData[0].idle=1 OR wData[1].idle=1 OR wData[2].idle=1 OR
400           wData[3].idle=1 OR wData[4].idle=1);
401     brk:=0; j:=0;
402     while(j<5 AND brk=0)
403     {if(wData[j].idle=1)then{brk:=1}else{j:=j+1}};
404     wData[j].idle:=2;
405     wnumber[tNum]:= 0;
406     MulRecord(j, valueA, valueB, row, col);
407     rKeys[i].idle:=1;
408     wData[j].idle:=3
409   }
410 )
411 };
412
413 function Reducer1(int tNum)
414 {
415   while(error=0 AND (rKeys[0].idle=2 OR rKeys[1].idle=2 OR
416     rKeys[2].idle=2 OR rKeys[3].idle=2 OR rKeys[4].idle=2 OR readOver=1))
417   {RecordReducer1(tNum)}
418 };
419
420 function WriteRecord1()
421 {
422   frame(fp, i, brk) and
423   (
424     FILE *fp<==fopen("reducer1.txt", "w") and int i, brk and skip;
425     if(fp=NULL)then
426     {error:=1 and output("reducer1.txt_cannot_be_opened.\n")}
427     else{
428       while(error=0 AND !(readOver=1 AND (rKeys[0].idle=1 AND
429         rKeys[1].idle=1 AND rKeys[2].idle=1 AND rKeys[3].idle=1 AND
430         rKeys[4].idle=1) AND (wData[0].idle=1 AND wData[1].idle=1 AND
431         wData[2].idle=1 AND wData[3].idle=1 AND wData[4].idle=1)))
432       {
433         i:=0; brk:=0;

```

```

434     await(wData[0].idle=3 OR wData[1].idle=3 OR wData[2].idle=3 OR
435           wData[3].idle=3 OR wData[4].idle=3);
436     while(i<5 AND brk=0)
437     {if(wData[i].idle=3)then{brk:=1}else{i:=i+1}};
438     fputs(wData[i].data,fp) and skip;
439     wData[i].idle:=1
440   };
441   fclose(fp) and skip
442 }
443 )
444 };
445 function ReadRecord2()
446 {
447   frame(fp,i, brk) and
448   (
449     FILE *fp<= fopen("reducel.txt", "r" ) and skip;
450     if(NULL=fp)
451     then(error:=1 and output("reducel.txt_cannot_be_opened.\n"));
452     else
453     {
454       while(feof(fp)=0)
455       {
456         await(rRec[0].idle=1 OR rRec[1].idle=1 OR rRec[2].idle=1 OR
457               rRec[3].idle=1 OR rRec[4].idle=1);
458         int i<=0 and int brk<=0 and skip;
459         while(i<5 AND brk=0)
460         {if(rRec[i].idle=1)then {brk:=1}else{i:=i+1}};
461         if(rRec[i].data=NULL)then
462         {rRec[i].data:=(char*)malloc(bufferSize* sizeof(char));};
463         fgets(rRec[i].data,bufferSize, fp) and skip;
464         if(rRec[i].data[0]>'0' AND rRec[i].data[0]<='9')then
465         {rRec[i].idle:=2}
466       };
467       readOver:=1
468       fclose(fp) and skip
469     }
470   )
471 };
472 function Reducer2(int Nnum)
473 {
474   while(!(readOver=1 AND (rRec[0].idle!=2 AND rRec[1].idle!=2 AND
475                           rRec[2].idle!=2 AND rRec[3].idle!=2 AND rRec[4].idle!=2)))
476   { RecordReducer2(tNum) }
477 };
478
479 function RecordReducer2(int tNum)
480 {
481   frame(key1,key2,token,buf,brk, i,j,en_val) and
482   (
483     BakeryAlg(tNum,rnumber ,rchoosing);
484     await(rRec[0].idle=2 OR rRec[1].idle=2 OR rRec[2].idle=2 OR
485           rRec[3].idle=2 OR rRec[4].idle=2 OR read_over=1);
486     if(readOver=1 AND (rRec[0].idle!=2 AND rRec[1].idle!=2 AND
487                       rRec[2].idle!=2 AND rRec[3].idle!=2 AND rRec[4].idle!=2))
488     then{rnumber[tNum]:=0}
489     else{
490       char *key1, *key2, *token[2] and int en_val<=0 and skip;
491       int i<=0, j<=0, brk<=0 and skip;
492       while(i<5 AND brk=0)
493       {if(rRec[i].idle=2)then{brk:=1}else{i:=i+1}};
494       rRec[i].idle:=3;
495       rnumber[tNum]:=0;
496       key1:=strtok(rRec[i].data, "_" );
497       key2:=strtok(NULL,"_");
498       token[1]:=key1;
499       while(NULL!=token[1])
500       {
501         token[0]:=strtok(NULL,"_");

```

```

502     en_val:=en_val+atoi(token[0]);
503     token[1]:=strtok(NULL,"_");
504     token[1]:=strtok(NULL,"_");
505     token[1]:=strtok(NULL,"_");
506 };
507 BakeryAlg(tNum,wnumber,wchoosing);
508 char buf[10] and j<=0 and brk<=0 and skip;
509 await(wData[0].idle=1 OR wData[1].idle=1 OR wData[2].idle=1 OR
510        wData[3].idle=1 OR wData[4].idle=1);
511 while(j<5 AND brk=0)
512 {if(wData[j].idle=1)then{brk:=1}else{j:=j+1}};
513 wData[j].idle:=2;
514 wnumber[tNum]:=0;
515 if(wData[j].data=NULL)then
516 {wData[j].data:=(char*)malloc(bufferSize* sizeof(char));
517  strcpy(wData[j].data, "") and skip;
518  strcat(wData[j].data,key1) and skip;
519  strcat(wData[j].data, "_" ) and skip;
520  strcat(wData[j].data, key2) and skip;
521  strcat(wData[j].data, ":") and skip;
522  itoa( en_val, buf, 10 ) and skip;
523  strcat(wData[j].data, buf) and skip;
524  strcat(wData[j].data, "\n" ) and skip;
525  rRec[i].idle:=1;
526  wData[j].idle:=3
527 }
528 )
529 };
530 function WriteRecord2()
531 {
532   frame(fp, i, brk) and
533   (
534     FILE *fp<==fopen("ProductMatrix.txt","w") and int i, brk and skip;
535     while(!(readOver=1 AND (rRec[0].idle=1 AND rRec[1].idle=1 AND
536                            rRec[2].idle=1 AND rRec[3].idle=1 AND rRec[4].idle=1) AND
537           (wData[0].idle=1 AND wData[1].idle=1 AND wData[2].idle=1 AND
538           wData[3].idle=1 AND wData[4].idle=1)))
539     {
540       i:=0; brk:=0;
541       await(wData[0].idle=3 OR wData[1].idle=3 OR wData[2].idle=3 OR
542            wData[3].idle=3 OR wData[4].idle=3);
543       while(i<5 AND brk=0)
544       {if(wData[i].idle=3)then{brk:=1}else{i:=i+1}};
545       fputs(wData[i].data,fp) and skip;
546       wData[i].idle:=1
547     };
548     fclose(fp) and skip
549   )
550 };
551
552 frame(rData,i,curRow,matrixID,readOver,wData,rnumber,rchoosing,wnumber,
553        wchoosing,rowA,colA,rowB,colB,error,colDistrA,rowDistrB,wColA,
554        wRowB,curKey,rKeys,bufferSize,rRec) and
555 (
556   ReadData rData[5] and Record wData[5] and int readOver<=0
557   and int rnumber[5] and int rchoosing[5] and int wnumber[5] and
558   int wchoosing[5] and int curRow<=1 and char matrixID and int i<=0
559   and int rowA<=0, colA<=0, rowB<=0, colB<=0, error<=0 and
560   int *colDistrA, *rowDistrB, *wColA, *wRowB and int curKey<=1 and
561   ReadRecord rRec[5] and int bufferSize<=1000 and Record rRec[5]
562   and skip;
563   while(i<5)
564   {
565     rKeys[i].idle:=1;rKeys[i].dataA:=NULL;rKeys[i].dataB:=NULL;
566     rData[i].idle:=1;rData[i].data:=NULL;
567     wData[i].idle:=1; wData[i].data:=NULL;
568     rRec[i].idle:=1 ;rRec[i].data:=NULL;
569     rchoosing[i]:=0; rnumber[i]:=0;

```

```

570     wchoosing[i]:=0; wnumber[i]:=0;
571     i:=i+1
572 };
573 (ReadMatrix() || HandleMatrix(0) || HandleMatrix(1) || HandleMatrix(2) ||
574   HandleMatrix(3) || HandleMatrix(4) || WriteMatrix());
575 if(error=1) then{output("error!") and skip}
576 else
577 {
578   colDistrA:=(int*)malloc(colA* sizeof(int));
579   rowDistrB:=(int*)malloc(rowB* sizeof(int));
580   wColA:=(int*)malloc(colA* sizeof(int));
581   wRowB:=(int*)malloc(rowB* sizeof(int));
582   i:=0;
583   while(i<colA)
584   {
585     colDistrA[i]:=0; rowDistrB[i]:=0;
586     wColA[i]:=0; wRowB[i]:=0; i:=i+1
587   };
588   Map1();
589   free(colDistrA) and free(rowDistrB) and
590   free(wColA) and free(wRowB) and skip;
591   readOver:=0;
592   (ReadRecord1() || Reducer1(0) || Reducer1(1) || Reducer1(2) || Reducer1(3)
593     || Reducer1(4) || WriteRecord1());
594   Map2();
595   readOver:=0;
596   (ReadRecord2() || Reducer2(0) || Reducer2(1) || Reducer2(2) || Reducer2(3)
597     || Reducer2(4) || WriteRecord2());
598   i:=0;
599   while(i<5)
600   {
601     if(rKeys[i].dataA!=NULL) then{free(rKeys[i].dataA) and skip};
602     if(rKeys[i].dataB!=NULL) then{free(rKeys[i].dataB) and skip};
603     if(rData[i].data!=NULL) then{free(rData[i].data) and skip};
604     if(wData[i].data!=NULL) then{free(wData[i].data) and skip};
605     if(rRec[i].data!=NULL) then{free(rRec[i].data) and skip};
606     i:=i+1
607   };
608 }
609 )

```

## References

1. <http://hadoop.apache.org/>
2. <https://storm.apache.org/>
3. <https://spark.apache.org/>
4. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A view of cloud computing. *Commun. ACM* **53**(4), 50–58 (2010)
5. Borthakur, D.: The hadoop distributed file system: architecture and design. *Hadoop Proj. Website* **11**(2007), 21 (2007)
6. Bryant, R.E.: Data intensive scalable computing. Carnegie Mellon University (2008). Accessed 10 Aug 2009
7. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pp. 1–6. ACM (1987)
8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. *Commun. ACM* **51**, 107–113 (2008)



9. Duan, Z.: An extended interval temporal logic and a framing technique for temporal logic programming. Ph.D. thesis, University of Newcastle upon Tyne (1996)
10. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Marri-  
rickville (2005)
11. Duan, Z., Tian, C.: A unified model checking approach with projection temporal  
logic. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp.  
167–186. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-88194-  
0\\_12](https://doi.org/10.1007/978-3-540-88194-0_12)
12. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier, Ams-  
terdam (2008)
13. Ihaka, R., Gentleman, R.: R: a language for data analysis and graphics. *J. Comput.  
Graph. Stat.* **5**(3), 299–314 (1996)
14. John Walker, S.: Big Data: A Revolution that Will Transform how We Live, Work,  
and Think (2014)
15. McAfee, A., Brynjolfsson, E., et al.: Big data: the management revolution. *Harvard  
Bus. Rev.* **90**(10), 60–68 (2012)
16. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R.,  
Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache hadoop yarn: yet another  
resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Com-  
puting, p. 5. ACM (2013)
17. Vora, M.N.: Hadoop-HBase for large-scale data. In: 2011 International Conference  
on Computer Science and Network Technology (ICCSNT), vol. 1, pp. 601–605.  
IEEE (2011)
18. Wang, M., Tian, C., Duan, Z.: Full regular temporal property verification as  
dynamic program execution. In: Proceedings of the 39th International Conference  
on Software Engineering Companion, pp. 226–228. IEEE Press (2017)
19. Wang, X., Tian, C., Duan, Z., Zhao, L.: MSVL: a typed language for temporal  
logic programming. *Front. Comput. Sci.* **11**(5), 762–785 (2017)
20. Yang, K., Duan, Z., Tian, C., Zhang, N.: A compiler for MSVL and its applications.  
*Theor. Comput. Sci.* (2017). <https://doi.org/10.1016/j.tcs.2017.07.032>
21. Yang, W., Liu, X., Zhang, L., Yang, L.T.: Big data real-time processing based on  
storm. In: 12th IEEE International Conference on Trust, Security and Privacy in  
Computing and Communications (TrustCom) 2013, pp. 1784–1787. IEEE (2013)
22. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster  
computing with working sets. *HotCloud* **10**(10–10), 95 (2010)
23. Zhang, N., Duan, Z., Tian, C.: A cylinder computation model for many-core parallel  
computing. *Theoret. Comput. Sci.* **497**, 68–83 (2013)
24. Zhang, N., Duan, Z., Tian, C.: A mechanism of function calls in MSVL. *Theoret.  
Comput. Sci.* **654**, 11–25 (2016)
25. Zhang, N., Duan, Z., Tian, C.: Model checking concurrent systems with MSVL.  
*Sci. China Inf. Sci.* **59**, 118101 (2016)
26. Zikopoulos, P., Eaton, C., et al.: Understanding Big Data: Analytics for Enterprise  
Class Hadoop and Streaming Data. McGraw-Hill Osborne Media (2011)

# **Verification and Validation**



# A Software Tool to Support the “Vibration” Method

Pan Zhao<sup>1(✉)</sup> and Shaoying Liu<sup>2(✉)</sup>

<sup>1</sup> Graduate School of Computer and Information Sciences,  
Hosei University, Tokyo, Japan  
panan0093@sina.com

<sup>2</sup> Faculty of Computer and Information Sciences,  
Hosei University, Tokyo, Japan  
sliu@hosei.ac.jp

**Abstract.** The “Vibration” method proposed in the literature offers a strategy for generating test data from an atomic predicate in a specification with the aim of achieving full path coverage in the program that implements the function defined by the predicate. However, how to efficiently generate adequate test data using the method from the same predicate to quickly traverse all of the related paths in the program is still an open problem. In this paper, we describe a prototype software tool we have built recently that supports the test data generation based on the principle of the “Vibration” method and an experiment to evaluate the effectiveness of the “Vibration” method supported by the tool.

**Keywords:** Specification · Vibration testing · Predicate-based test generation

## 1 Introduction

Formal specification-based testing offers a rigorous and systematic approach to test data generation and test result analysis [3, 10]. In many cases, it can be automatically performed. In particular, it is superior to conventional functional testing techniques in automatically deriving test oracle that can be used to automatically determine whether a test finds bugs or not.

However, there are still many challenges as stated by Liu and Nakajima in [8]. One of them is how to generate adequate test data from a formal specification that covers all of the execution paths in the corresponding program. To tackle this problem, Liu and Nakajima proposed a special test data generation method based on predicates, called “Vibration” method (or V-Method), that is expected to efficiently generate adequate test data to cover all of the corresponding paths in the program [1]. A small experiment has shown that the V-Method is effective in achieving the path coverage.

To enable the V-Method to be automatically applied in test data generation, tool support is necessary. But unfortunately, no software tool for V-Method has been built so far. In this paper, we report a prototype tool for V-Method that we have recently developed. The purpose is to facilitate users of the tool (or testers) to investigate the relation between the important elements such as the vibration “distance” used in the V-Method algorithm [1] and the path-coverage in the program. Apart from this major

contribution, we also discuss how our tool is constructed by first formally specifying the desired functionality using the SOFL specification language [2] and then implementing it with C#. An experiment is conducted to investigate how effective the V-Method is in terms of path-coverage in the program and by comparing with the pairwise testing [6].

The remainder of this paper is organized as follows: Sect. 2 gives a brief introduction to the SOFL specification and functional scenario-based testing. Section 3 introduces the principle of the V-Method and how it works specifically. Section 4 explains the functions of the supporting tool and the development process. In Sect. 5, three experiments and their analysis are presented. Section 6 makes some discussions and conclusions according to the research. Section 7 discusses the weakness of the present work and how it can be improved in the future.

## 2 SOFL Specification and Functional Scenario-Based Testing

SOFL (Structured Object-oriented Formal Language) is one of the formal engineering methods for industrial software development. It is characterized by integrating the advantages of Data Flow Diagrams, Petri nets, and VDM-SL (Vienna Development Method-Specification Language) [2]. The traditional data flow diagram (DFD) is formalized into condition data flow diagram (CDFD) by using Petri nets to provide an operational semantics for DFD. Each CDFD is associated with a module in which all of the components of the CDFD, such as processes, data flows, and data stores, are precisely defined with a formal notation. CDFDs and their associated modules are organized in a hierarchy formed by decomposing high level processes into sub-CDFDs. Compared with other formal methods, the syntax of SOFL is much simpler and therefore it can be used to easily write comprehensible formal specifications. Since our discussion in this paper only needs the knowledge of a process specification in SOFL, we merely focus on the introduction of SOFL process specification in this section.

The following is an example of process specification:

```
process A (x: int) y: int
pre    x > 0
post  x <= 10 and y = 1 / x or
      x > 10 and y = x * 5

end_process
```

This detailed specification can be represented by an abstract version S (Siv, Sov) [Spre, Spost] to facilitate our discussion on issues in relation to test data generation. In the abstract representation, Siv denotes the set of input variables and Sov the set of output variables, and the pre- and post- conditions define the functionality of the process in terms of the relation between input and output.

In the second author's previous research, a concept of functional scenario was proposed [5]. A set of functional scenarios can be derived from a formal specification, each defining an independent function in terms of input-output relation.

Let  $S_{post} = (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee \dots \vee (G_n \wedge D_n)$ , where  $G_i$  is a guard condition and  $D_i$  a defining condition, and  $i = 1, \dots, n$ . Then, a functional scenario form (FSF) of  $S$  is defined as  $(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee \dots \vee (S_{pre} \wedge G_n \wedge D_n)$ , where  $S_{pre} \wedge G_i \wedge D_i$  is called functional scenario. For instance, a set of functional scenarios derived from process  $A$  given above is as follows:

1.  $x > 0 \wedge x \leq 10 \wedge y = 1/x$
2.  $x > 0 \wedge x > 10 \wedge y = x * 5$
3.  $x \leq 0 \wedge \text{Anything}$

A distinct characteristic of a functional scenario is its independence in defining a specific relation between input and output. Taking the advantage of this characteristic, a *scenario-based testing method* has been proposed in [8, 9]. The central idea of this method is: generate test data for each functional scenario at least one by satisfying each atomic predicate of the *test condition* that is the conjunction of the pre-condition and the guard condition ( $S_{pre} \wedge G_i$ ) in a scenario. We have already built algorithms for automatic test data generation based on atomic predicates [8], but those algorithms can only produce one test data for each functional scenario at a time. Since a functional scenario is usually refined into a collection of program paths, only one test data is obviously insufficient to cover all of the related paths. To address this problem, on the basis of the scenario-based testing, the *V-Method* for generating test data was put forward in [1].

### 3 Introduction to V-Method

#### 3.1 Principle of the V-Method

The purpose of the V-Method is to generate an appropriate test set automatically from an atomic predicate, which is usually a relation, with the aim of attempting to traverse all of the representative paths in the program. In this part, we will give a brief introduction to the principle of the V-Method.

Let an atomic predicate from a functional scenario be  $E_1(x_1, x_2, \dots, x_n) R E_2(x_1, x_2, \dots, x_n)$ , where  $R$  is a relational operator, such as  $>$ ,  $<$ ,  $\geq$ , or  $\leq$ , and  $E_1$  and  $E_2$  are expressions that contain all the input variables  $x_1, x_2, \dots, x_n$ . Following the V-Method to generate test data from the atomic predicate, we first generate a test data containing specific values for all of the input variables  $x_1, x_2, \dots, x_n$  that satisfy the predicate. Then, we will generate another test data that also satisfies the predicate, but ensures that the “distance” between the two expressions  $E_1$  and  $E_2$  meets a certain given condition. In the case of both  $E_1$  and  $E_2$  being numeric expressions, the distance between them is defined as absolute value of  $E_1 - E_2$ . If necessary, we can repeatedly change the distance according to the certain principle by increasing or decreasing it properly and then generate more test data to satisfy the predicate. The test data generated in this way are expected to efficiently cover the related paths in the program.

An algorithm to realize the V-Method was developed by the second author previously in [1]. We quote the algorithm in the Appendix in order to help the reader

understand better our discussions where some variables and concepts involved in the algorithm may be used. The essential idea of the algorithm has been explained briefly previously. Readers can refer to publication [1] for a detailed explanation.

### 3.2 Example

We give a specific example to illustrate how the general principle of the V-Method works.

Let  $x + 5 > y \wedge z = x + y$  be a functional scenario where  $x + 5 > y$  is an atomic predicate describing a test condition of the functional scenario and  $z = x + y$  is the defining condition showing how the output variable  $z$  is defined using input variables  $x$  and  $y$ . Since the test condition contains only input variables, it is used as the basis for test data generation. To begin with, an initial test data is generated to satisfy the predicate  $x + 5 > y$ , for example,  $x = 3$  and  $y = 2$ . Then, a “vibration” process of repeatedly generating test data to satisfy both the predicate and the required distance (e.g., 8) between  $x + 5$  and  $y$  starts. To control the process of the “vibration”, we need to determine several parameters used in the test data generation algorithm given in the Appendix, such as “distance\_up” (e.g., distance\_up = 6) and “distance\_down” (e.g., distance\_down = 3). In general, the value of distance\_up should be greater than that of distance\_down in order to allow as many times of “vibration” as necessary. For instance, to generate the next test data with the above values of the parameters, we need to generate a test data, say  $x = 5$  and  $y = 2$ , to meet the atomic predicate  $x + 5 - y = 8$  (ensuring the distance between the two expressions  $x + 5$  and  $y$  is 8). Then, we can increase the distance by adding the value 6 of distance\_up to the value 8 of distance to update the distance to value 14. Then, we can generate another test data to meet the predicate  $x + 5 - y = 14$ , say  $x = 12$  and  $y = 3$ . After this, we need to shorten the distance by reducing the value 3 of distance\_down from the current value 14 of distance and use the result to update the distance (e.g., making distance = 11). Then, we can generate another test data, say  $x = 7$  and  $y = 1$ , to meet the atomic predicate  $x + 5 - y = 11$ . Repeating this process, more necessary test data can be generated.

When the V-Method is used to generate test data manually, the values of all the relevant parameters will be determined by humans. But if it is used to automatically generate test data, the values of the parameters must be automatically decided. But how to automatically make the decision to achieve pre-defined goals still remains an open problem to be addressed. In this paper, we still cannot attack this problem due to the lack of sufficient research, but focus on the software tool support as our contribution.

## 4 Supporting Tool for “Vibration” Method

We have built a prototype software tool, known as “Vibration Testing Tool” (VTT), to support the V-Method using C# in the Visual Studio 2015 environment. The user of the tool only needs to input the necessary initial values, click the button, and then the tool

can automatically generate test data based on the V-Method. Figure 1 shows a snapshot of the graphical user interface of the tool. It contains two parts: the upper-left corner and the bottom-right corner are two function modules which provide the input bars for users; the rest of the tool is output function part, including chart expression (showing the changes of distance), the path coverage number, and the show boxes of the test data generated.

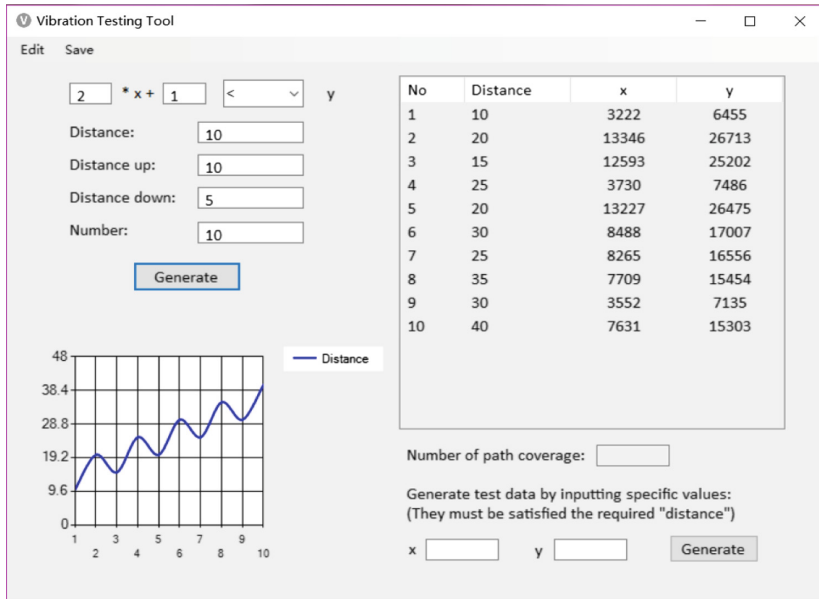


Fig. 1. GUI of VTT

The functions of the tool are designed using Condition Data Flow Diagrams (CDFDs) together with the related modules of the SOFL specification language, which are explained next.

### 4.1 Functions

Figure 2 is the CDFD defining the tool functions at an abstract level, which is drawn with the SOFL Toolkits as reported in [11]. As the CDFD shows, the tool performs four specific functions, but only two major functions are worth explaining in detail.

**Receiving Initial Parameter Values and Produce Distance Values.** As discussed previously, there are several parameters used in the V-Method algorithm for test data generation, such as distance, distance\_up, distance\_down, and number (the number of test data to be generated). To use the V-Method, the tool first requires the user to supply

the initial values for all these parameters. Then, the tool will automatically produce a set of distance values based on the initial values of the parameters for executing the V-Method algorithm to automatically generate test set (a set of test data). The snapshot in Fig. 1 shows a situation where all of the parameters are given an initial value and a set of test data are automatically generated accordingly.

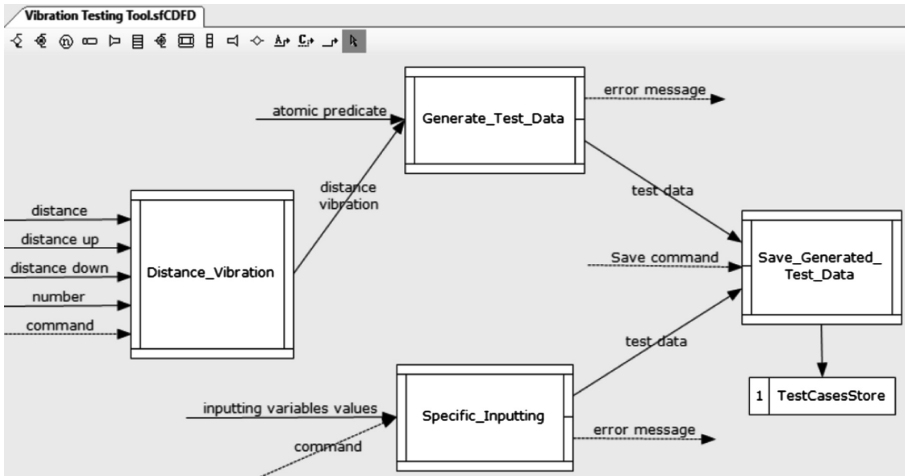


Fig. 2. CDFD of VTT

**Automatically Generating Test Data.** This function first receives an atomic predicate, for example  $ax + b > y$  (linear relation), where  $a$  and  $b$  are two constants, and  $x$  and  $y$  are input variables. The relational operator can be any of the following: =, <> (inequality), <, >, ≤ (less than or equal to), and ≥ (greater than or equal to). Under the condition that both the atomic predicate and the distance values produced previously are satisfied, the tool will automatically generate the pre-defined number of test data for the input variables. The tool also supports the interactive test data generation. In this case, the test data are supplied manually by the user through the GUI, but whether they satisfy the required condition of the V-Method will be automatically checked. If the condition is violated, an appropriate error message will be displayed to remind the user of what is wrong in his or her data.

#### 4.2 Quality Assurance of the Tool Using SOFL

The quality of the tool itself is closely related to the quality of test data generation and the enforcement of the V-Method. We have adopted the SOFL three-step specification approach [2] first to construct a formal specification for the tool and then implements it based on the specification. In this sense, our work demonstrates another case of applying SOFL in practice.



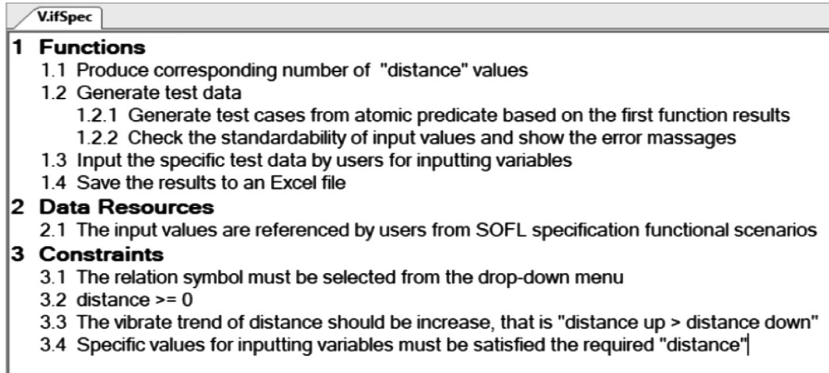


Fig. 3. Informal specification of VTT

Following the SOFL three-step approach, we started with the construction of an informal specification, abstractly describing the desired functions, data resources necessary for realizing the functions, and constraints either on the functions or the data resources. Figure 3 shows the informal specification of VTT.

On the basis of the informal specification, we then built a semi-formal specification by doing three things. One is to group the related functions, data resources, and constraints given in the informal specification into SOFL modules. Another activity is to precisely define all of the necessary types and declare state variables for each module. These types and variables are expected to be used in specifying the functionality of related operations which we call processes in SOFL specifications. The final activity is to specify the functionality of all of the related processes in each module using pre- and post-conditions. While all of the data items of each process are formally declared with well-defined types, the pre- and post-conditions are given informally, usually in natural language such as English. The specification for the function *Distance\_Vibration* given below characterizes the semi-formal specification of VTT.

**function** *Distance\_Vibration* (*distance*: real, *distance\_up*: real, *distance\_down*: real, *number*: nat, *command*: bool) : seq of distance

**pre** *The input values are available*

**post** *On the basis of the previous distance value, the operation is added or reduced to get a new distance value, and each distance value is stored in a sequence*

**end\_function**

While the semi-formal specification rather clearly defines the functions of the tool, many expressions inside are still given informally and contain ambiguities. To form a solid foundation for implementation, we have refined the semi-formal specification into a formal specification in which all parts are described in the SOFL formal specification language. Below is a module in the formal specification of VTT, which contains two processes *Generate\_Test\_data* and *Specific\_Inputting*, one function *Distance\_Vibration*, and the necessary type and variable declarations. For brevity, we omit further explanation of the specification.

**Module** *Vibration\_Testing\_Tool*

**type** *distance* = real;

*change\_value* = real;

*test\_case* = real;

**var***x*: seq of *test\_case*;

*y*: seq of *test\_case*;

*test\_data*: seq of *test\_case*;

*test\_data\_store*: seq of *test\_case*;

**behav**: CDFD\_Fig. 3;

**process** *Generate\_Test\_data* (*atomic\_predicate*: bool, *Distance\_Vibration*: seq of distance) *test\_data*: seq of *test\_case*, *msg*: string

**post** if *Distance\_Vibration* = null

then *msg* = "The input values are not available!"

else forall [*i*: inds (*Distance\_Vibration*)] forall [*Distance\_Vibration* (*i*) : distance] exists [*x* (*i*) : *test\_case*, *y* (*i*) : *test\_case*] *atomic\_predicate* = true and *test\_data* = conc (*x*, *y*)

**end\_process**

**process** *Specific\_Inputting* (*atomic\_predicate*: bool, *Distance\_Vibration*: seq of distance, *variable\_1*: real, *variable\_2*: real) *test\_data*: seq of *test\_case*

**pre** exists [*i*: inds (*Distance\_Vibration*)] *atomic\_predicate* = true

**post** *test\_data* = [*variable\_1*, *variable\_2*]

**process** *Save\_Generated\_Test\_Data* (*test\_data*: seq of *test\_case*)

**ext** wr *test\_data\_store*

**post** *test\_data\_store* = conc (~*test\_data\_store*, [*test\_data*])

**end\_process**

**function** *Distance\_Vibration* (*distance*: real, *distance\_up*: real, *distance\_down*: real, *number*: nat, *command*: bool) : seq of distance

**pre** *distance* >= 0 and *distance\_up* > *distance\_down*

**post** if *command* = true then *change\_value* = *distance\_up* and *command* = false

else *change\_value* = -*distance\_down* and *command* = true

if *distance* = *distance* + *number* div 2 \* *distance\_up* - (*number*-1) div 2 \* *distance\_down*

then *Distance\_Vibration* = [*distance*]

else *Distance\_Vibration* = conc (*distance*, *Distance\_Vibration* (*distance* + *change\_value*, *distance\_up*, *distance\_down*, *number*, *command*))

**end\_function**

**end\_Module**

## 5 Experiments

Three experiments, two small ones and one relatively big one in terms of the scale of the program used for testing, are conducted for evaluation of the V-Method by comparing it with the "Pairwise testing" available in the literature. The reason we use "Pairwise testing" for the comparison is that it is a popular technique for test data generation in practice [6], often used in industry [7], and suitable for automation [1].

The programs used for testing in the experiments are chosen from different application domains in order to avoid biases and perform an objective evaluation. The role of our tool VTT in the experiments focuses on automatic test data generation using the V-Method and manual test data generation using the pairwise testing method, as well as executing the target programs to report the path-coverage. These experiments also help us demonstrate the practicability, feasibility, and efficiency of our tool VTT.

### 5.1 BMI (Body Mass Index) Calculation

BMI is a commonly used measure standard for measuring the degree of obesity (5 levels) and whether a person is healthy or not, and it is a relatively objective parameter by body weight and stature. The formal specification of the BMI calculation process and the reference standard (divided into five paths) are (Table 1):

```
process Adult_BMI_calculation (stature: real, weight: real) BMI: real
pre  stature > 140 and stature < 220 and weight > 35 and weight < 150
post weight > 0.63 * stature - 66.15 and BMI = weight / (stature * stature / 10000)
end_process
```

**Table 1.** BMI reference standard

1	BMI < 18.5	Underweight
2	18.5 ≤ BMI < 23	Normal
3	23 ≤ BMI < 25	Overweight
4	25 ≤ BMI < 30	Obese
5	BMI ≥ 30	Clinically obese

In this case, the guard condition  $weight > 0.63 * stature - 66.15$  is the atomic predicate relation that will be used in generating test data. This relation is figured out by the body weight standard formula and actual situation, which requires that the weight of a human body not be below a certain value. Based on the pre- and post-conditions and come historical data, we derive the initial values for all of the parameters in the V-Method algorithm as  $distance = 13$ ,  $distance_{up} = 15$ ,  $distance_{down} = 3$ , and  $number = 5$ . Using these input values, VTT generates 5 test data that cover all of the paths in the program implementing the specification. That is, each change of the distance value makes the test data cover a new path. A snapshot of the GUI for this experiment is shown in Fig. 4. The input variable  $x$  denotes the stature and  $y$  the weight, and for this example, we add two variables  $z$  and  $path$  in the tool to show the results of the V-Method more intuitively, where  $z$  is the calculated BMI and  $path$  visualizes the traversed path number by the corresponding test data. The chart at the bottom-left of the GUI shows the changing of the distance values as the number of the test data changes from 1 to 5.

On the other hand, we generate the same number of test data using the pairwise testing. Specifically, we choose one value for the variable  $stature$  between 140 and 220 (possible range suitable for humans), and 5 values for the variable  $weight$  between 35 and 150 (reasonable range for most people) to produce  $1 * 5 (= 5)$  test data.

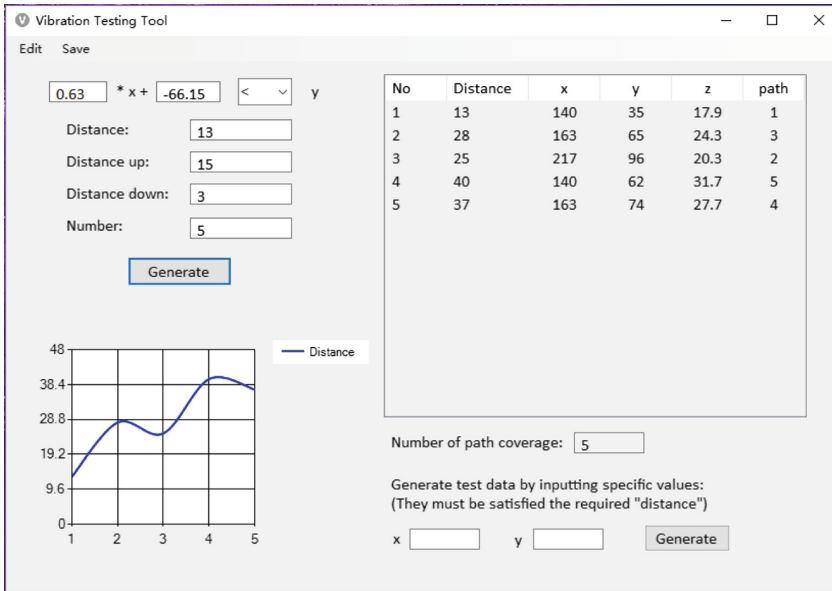


Fig. 4. Test data generation for BMI calculation

### 5.2 Balance Ratio Calculation

Balance ratio refers to the ratio of the balance and income of a family in a certain period. It reflects the ability and saving awareness of the family to control spending, and is the basis for future investment and financing. This function is defined in a SOFL process specification given below.

*process* Balance\_Ratio\_Calculation(*income: real, expenditure: real*)

*balance\_ratio: real*

*pre*  $income > 1000$  and  $income < 20000$  and  $expenditure > 0$

*post*  $income \geq expenditure$  and  $balance\_ratio = (income - expenditure) / income$

*end\_process*

Income and expenditure are the input variables, balance ratio is the output. The guard condition in the only functional scenario of the post-condition is  $income \geq expenditure$ .

On the basis of this predicate, a number of test data are produced. Figure 5 shows a snapshot of the GUI for producing 12 test data for this process. where  $x$ ,  $y$ ,  $z$  and  $path$  denote “income”, “expenditure”, “balance ratio” and “the covered path number”, respectively.

Meanwhile, we also generate 12 test data using the pairwise testing, which is the same as that of test data generated using the V-Method. Specifically, we evenly choose three different values for the variable *income* between 1000 and 20000 (reasonable

range of monthly income for people in China), and 4 values for the variable *expenditure* between 0 and 20000 (reasonable range for average people) to produce  $3 * 4 (= 12)$  test data.

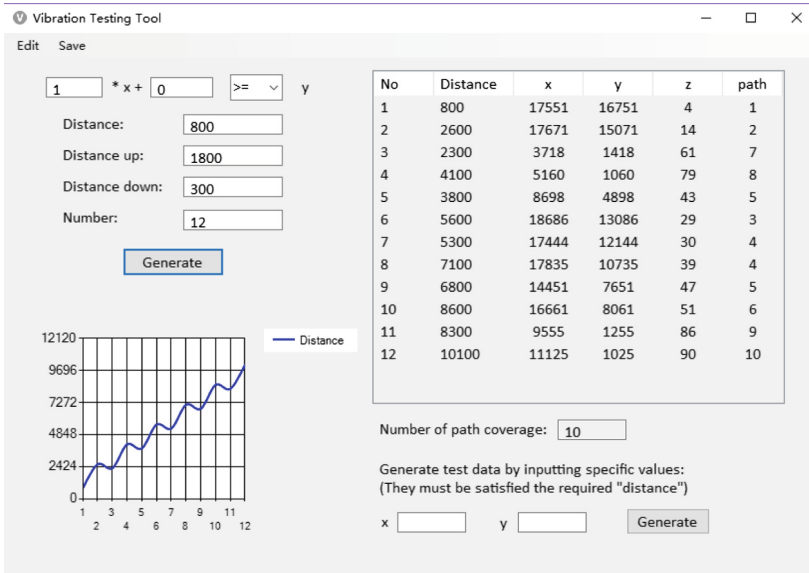


Fig. 5. Test data generation for Balance ratio calculation

### 5.3 Train Fare System

The train fare system is used to calculate the cost for passengers’ tickets based on their starting station and arrival station. It adopts fifty-seven stations in China. The calculation of the costs for train tickets based on the starting station number and the arrival station number contains 1596 paths. The specification for the calculation is as follows:

**module** *Train\_Fare\_System*

**type** *starting* = nat;

*arrival* = nat;

*fare* = real;

*Fare\_schedule* = map *starting* and *arrival* to *fare*;

**process** *Train\_Fare\_Calculation* (*starting*: real, *arrival*) *fare*: real

**ext** rd *fare\_schedule\_file*: *Fare\_schedule*

**pre** *starting* < *arrival*

**post** *arrival* - *starting* <= 5 and *fare* = *fare\_schedule\_file* (*starting*, *arrival*) or  
 5 < *arrival* - *starting* <= 10 and *fare* = *fare\_schedule\_file* (*starting*, *arrival*) or  
 10 < *arrival* - *starting* <= 30 and *fare* = *fare\_schedule\_file* (*starting*, *arrival*) or  
 30 < *arrival* - *starting* <= 50 and *fare* = *fare\_schedule\_file* (*starting*, *arrival*) or  
 50 < *arrival* - *starting* <= 56 and *fare* = *fare\_schedule\_file* (*starting*, *arrival*)

**end\_process**

In the specification, *starting* and *arrival* are the input variables and *fare* is the output variable. In the post-condition, there are five 5 functional scenarios. We have generated 2000 test data with the V-Method to cover 1318 paths in the program. Figure 6 illustrates the situation of using our tool VTT to generate test data (only part of them can be seen), where *x* and *y* denote input variables “starting” and “arrival”, respectively.

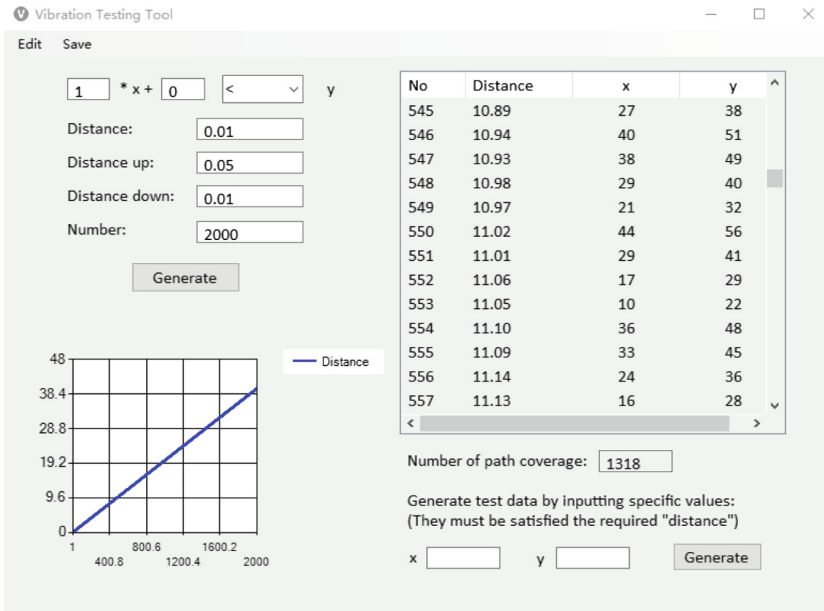


Fig. 6. Test data generation for Train fare system

In addition, we also generate 2000 test data using the pairwise testing, which is the same as that of test data generated using the V-Method in this case. Specifically, we choose 40 different values for the variable *start* between 1 and 55 (reasonable range of stations we adopt as a starting in the system), and 50 values for the variable *arrival* between 2 and 56 (reasonable range of stations that can be used as the arrival station) to produce  $40 * 50 (= 2000)$  test data.

### 5.4 Experiment Results and Analysis

To avoid biases in comparison with pairwise testing, we try to generate the equal number of test data from both the V-Method and the pairwise testing method. As we generate test data, we execute the program to check the path coverage. When all the test data generated so far from one method have reached 100% of path coverage, the generation of test data from both methods will terminate, and the number of the test data generated so far from each method is then treated as the final test data generated from that method. But when this strategy is impossible to achieve due to bugs in the

program, for example, in the case of the third experiment, we stop generating test data when the recently generated test data do not traverse new paths in the program.

Table 2 shows the summary of the test results for the three systems using the two test data generation methods. In terms of the effectiveness of achieving path-coverage, obviously the V-Method is superior to the pairwise testing in all of the three cases. But in terms of the number of test data used, the V-Method tends to use the same or more test data than the pairwise testing, although the difference between the numbers of test data generated using the two methods is not significant. However, considering the fact that all of the test data can be automatically generated using the V-Method, supplying more test data by the V-Method would not be a significant weakness. This little weakness may be compensated by its superiority in automatically deriving test oracle and analyzing test results based on it.

**Table 2.** Path coverage rate results

Programs (number of paths, number of test data)	Pairwise testing	V-Method
BMI calculation (5, 5)	80%	100%
Balance ratio calculation (10, 12)	50%	100%
Train toll system (1596, 2000)	61%	83%

## 6 Discussion

The decision on “distance” values is the core issue in the V-Method. If the program under test (PUT) contains a great number of paths, the “vibration” (or changing) of the distance value should be more frequently done and its amplitude should be sufficiently small so as to ensure more paths being traversed quickly. On the other hand, if the path distribution in the program is dispersed due to the fact that the distance between input variables is relatively big, then the distance vibration amplitude should sufficiently big and the vibration frequency should be sufficiently low, in order to avoid too many ineffective test data being produced.

It seems to be quite hard to provide a commonly recognized and universally effective distance vibration frequency and amplitude for the V-Method. In general, these values should be determined in accordance with the characteristics of the specific programs under testing. The quality of the distance values can be generally judged based on the fact of whether every newly produced test data after the distance value being adjusted can make more paths traversed if more untraversed paths still remain.

We must also mention the limitation of our work presented in this paper. Currently, the tool VTT can only support the automatic test data generation from relations between two numeric expressions, not other types of expressions, such as set types, sequence types, and map types. Therefore, our three experiments also use mere numeric relations and the results are apparently limited. The effectiveness of the V-Method with a more appropriate tool support will have to be investigated on broader types of predicates in the future.

## 7 Conclusion and Future Work

In this paper, we have described a software tool called VTT to support the V-Method available in the literature. The tool can be used in three steps: (1) determining the atomic predicate from a functional scenario of the specification; (2) supplying initial values for the parameters distance, distance\_up, distance\_down, and test data number based on the program under testing; (3) automatically generating test data. Apart from our contribution in building the tool, we have also carried out three experiments to evaluate the effectiveness of the V-Method by comparing it with the pairwise testing. The results show the superiority of the V-Method to the pairwise testing in all of the three cases, but this conclusion is limited to relations between numeric expressions.

Our future work will focus on the improvement of our tool VTT to enable it to deal with various types of non-numeric atomic predicates. Meanwhile, we will also try to establish a theory for determining distance values between two expressions in an atomic predicate. While this may be challenging in general, but the problem may be alleviated if the application domain is limited to a specific one. Moreover, we will also be interested in evaluating the V-Method on the basis of large-scale and complex industrial program systems.

**Acknowledgments.** This work was supported by JSPS KAKENHI Grant Number 26240008.

## Appendix

### Algorithm 1

No.1  $TraversedPathsSoFar := \{\};$   $Number := 0;$  /\* indicating how many times of generating test cases has been performed without increasing traversed paths. \*/

No.2  $tem\_distanceUp := distanceUp;$

$tem\_distanceDown := distanceDown;$

No.3 Generate a test case  $t = \{(x_1, v_1), (x_2, v_2), \dots, (x_m, v_m)\}$  using existing algorithm [8] such that  $E_1(v_1, v_2, \dots, v_n) R E_2(v_1, v_2, \dots, v_n)$  holds;

$d := Distance(E_1(v_1, v_2, \dots, v_n), E_2(v_1, v_2, \dots, v_n), R);$  /\* representing the distance between the evaluation results of the two expressions on the generated values of the input variables. \*/

$ap := E_1(v_1, v_2, \dots, v_n) R E_2(v_1, v_2, \dots, v_n);$

$TraversedPathsSoFar := TraversedPathsSoFar \cup \{TraversedPaths(ap, t)\};$  /\* collecting all the traversed paths by the current test case. \*/

No.4 if  $d < 0$

then if times = 0

then  $d_1 := d + tem\_distanceUp$

else  $d_1 := d + tem\_distanceDown$  /\* let  $d_1$  is a new distance between  $E_1$

and  $E_2$ . \*/

else  $d_1 := d$



No.5 Generate a test case  $t_1 = \{ (x_1, v_1), (x_2, v_2), \dots, (x_n, v_n) \}$  such that  $E_1(v_1, v_2, \dots, v_n) R E_2(v_1, v_2, \dots, v_n)$  holds and  $d \leq \text{Distance}(E_1(v_1, v_2, \dots, v_n), E_2(v_1, v_2, \dots, v_n), R)$ ; /\* calculating the distance between the two values. \*/

No.6 if  $\text{TraversedPaths}(ap, t_1) \in \text{TraversedPathsSoFar}$  and  $\text{Number} \leq 10$   
 then {  $\text{Number} := \text{Number} + 1$ ;  
 $d := d_1$ ;  
 $ap := E_1(v_1, v_2, \dots, v_n) R E_2(v_1, v_2, \dots, v_n)$ ;  
 $\text{times} := (\text{times} + 1) \bmod 2$ ; go to No.4;}  
 else if  $\text{Number} \leq 10$   
 then {  $\text{TraversedPathsSoFar} := \text{TraversedPathsSoFar} \cup \{\text{TraversedPaths}(ap, t_1)\}$ ;  
 $d := d_1$ ;  
 $ap := E_1(v_1, v_2, \dots, v_n) R E_2(v_1, v_2, \dots, v_n)$ ;  
 $\text{times} := (\text{times} + 1) \bmod 2$ ;  
 $\text{Number} := 0$ ; go to No.4; }  
 else go to No.7;

No.7 End.

## References

1. Liu, S., Nakajima, S.: A “vibration” method for automatically generating test cases based on formal specifications. In: 2011 18th Asia Pacific Software Engineering Conference, pp. 73–80, 5–8 December 2011
2. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer, Heidelberg (2004). <https://doi.org/10.1007/978-3-662-07287-5>
3. Hörcher, H.M.: Improving software tests using Z Specifications. In: Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1995. LNCS, vol. 967, pp. 152–166. Springer, Heidelberg (1995). [https://doi.org/10.1007/3-540-60271-2\\_118](https://doi.org/10.1007/3-540-60271-2_118)
4. Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In: France, R., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 416–429. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-46852-8\\_30](https://doi.org/10.1007/3-540-46852-8_30)
5. Liu, S., Chen, Y., Nagoya, F., McDermid, J.A.: Formal specification-based inspection for verification of programs. IEEE Trans. Softw. Eng. **9**(4), 1100–1122 (2012)
6. Tai, K.C., Lei, Y.: A test generation strategy for pairwise testing. IEEE Trans. Softw. Eng. **28**(1), 109–111 (2002)
7. Czerwonka, J.: Pairwise testing in real world. In: proceedings of 24th Pacific Northwest Software Quality Conference (2006)
8. 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
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. Donat, M.R.: Automating formal specification-based testing. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997. LNCS, vol. 1214, pp. 833–847. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0030644>
11. Li, M., Liu, S.: Integrating animation-based inspection into formal design specification construction for reliable software systems. *IEEE Trans. Reliab.* **65**(1), 88–106 (2016)



# A Software Tool to Support Scenario-Based Formal Specification for Error Prevention

Siyuan Li<sup>1</sup>(✉) and Shaoying Liu<sup>2</sup>(✉)

<sup>1</sup> Graduate School of Computer and Information Sciences,  
Hosei University, Tokyo, Japan

LiSiyuan20170724@outlook.com

<sup>2</sup> Faculty of Computer and Information Sciences, Hosei University,  
Tokyo, Japan

sliu@hosei.ac.jp

**Abstract.** Formal specification can be an error-prone process for complex systems and how to efficiently write correct specifications is still a challenge for practitioners in industry. This paper presents a software tool to support the scenario-based formal specification approach developed in the SOFL formal engineering method. Using the tool, the current version of the formal specification under construction can be automatically checked to ensure the internal consistency and some further contents of the specification may be automatically predicated to facilitate the user in completing the specification. To improve the readability of the formal specification, the tool can also automatically translate the textual format of the specification into a comprehensible tabular format. All of these three functions can be helpful to prevent errors during the construction of the specification. We discuss each of the functions by first presenting its principle and then illustrating it with examples. We present a case study to show how the tool supports the scenario-based specification approach. Finally, we conclude the paper and suggest topics for future research.

**Keywords:** SOFL · Formal specification · Verification · Error prevention

## 1 Introduction

As a formal engineering method for practical software development, SOFL has provided a method for functional scenario-based inspection and testing of programs, respectively [1]. The same concept has also been found to be effective in helping construct formal specifications in practice [2] and verify their properties such as consistency [3] and completeness [4].

However, writing a formal specification for a complex system tends to be error-prone. This problem can be attributed to three factors according to the second author's experience in applying the SOFL method to develop realistic software systems. The first is that the practitioner who writes the formal specification may lack competence of commanding the formal notation. The second is that the habit of writing code may affect the practitioner in keeping the logical consistency and completing the definition of the functionality. For example, when defining an update of a composite

object, only some fields of the object are defined in the post-condition while the other fields are undefined. This style may have no problem for programming but usually results in an incomplete functional definition in the specification. The final factor has something to do with inappropriate input or output variables and their types. Choosing inappropriate variables and types may lead to errors in specification in most cases.

To deal with the above challenge, we believe that dynamically checking the consistency of the current version of the specification and predicating the further necessary contents of the specification as it is being constructed are an efficient way to build correct formal specifications. To realize this goal, obviously a software tool needs to be developed to support the process of constructing a specification. In our work, we develop a tool to support the functional scenario-based formal specification approach based on the SOFL specification language. Specifically, the tool supports three major functions: (1) automatically checking the internal consistency of the current version of the formal specification under construction, (2) automatically predicating further necessary contents for the specification, and (3) automatically translating the textual format of the specification into a comprehensible tabular format. All of these three functions can be helpful to prevent errors during the construction of the specification. The result of this research is expected to make the SOFL method more effective and practical in industry where the current SOFL technology has been tested or applied in realistic systems development.

The rest of this paper is organized into six sections. Section 2 briefly introduces the essential idea of the scenario-based formal specification for a process using the SOFL specification language. Section 3 explains the principles in detail. Section 4 shows the structure of the tool, then gives some examples to explain how it works. Section 5 shows some related work in the field. Section 6 gives a discussion and concludes the paper. Finally, Sect. 6 gives conclusion of this paper, and summarizes some research directions for future work.

## 2 Scenario-Based Formal Specification

In this section, we briefly introduce the essential idea of the scenario-based formal specification for a process using the SOFL specification language [5]. To this end, we first need to explain the concepts of *process* and *functional scenario*, and then illustrate the scenario-based approach to formal specification with an example.

### 2.1 Process Specification

A process is the essential component of a module in a SOFL formal specification. Its specification is composed of the signature, pre- and post-conditions. The signature shows the process name, input variables, output variables, and external variables. The pre-condition sets a restriction on the input of the process while the post-condition defines the relation between the input and the output that must be satisfied after the

```

process Operate(x: real ) message: string
ext wr y: real
pre true
post x > 0 and y = 1 and message = "result1" or
      x = 0 and y = 0 and message = "result2" or
      x < 0 and y = -1 and message = "result3"
end_process

```

execution of the process. Below is an example showing the structure of a process specification.

All of the types used to declare the input, output, and external variables in the process must be clearly defined in the type section of the related module. A module is a mechanism for defining a sub-system by describing its architecture using a condition data flow diagram (CDFD) and specifying the functionality of every process occurring in the CDFD. It also allows necessary constant identifiers, type identifiers, store variables, type and store invariants to be defined properly, which can be used in process specifications. Figure 1 illustrates the general structure of a SOFL module that is divided into three parts. The first part is for declarations of necessary data items; the second part is a collection of the process specifications; and the third part defines some functions that may be applied in some process specifications.

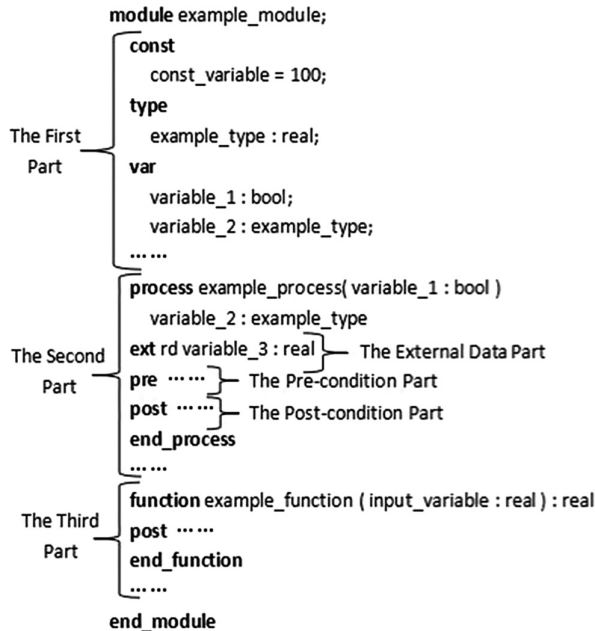


Fig. 1. Structure of a module.

## 2.2 Functional Scenarios

As mentioned in the Introduction, our collaboration with industry has helped us reveal the fact that a process specification can be effectively constructed by building the disjunction of *functional scenarios*. A functional scenario is a conjunction of the pre-condition, a *guard condition*, and a *defining condition*. The guard condition is defined as part of the post-condition and is characterized by including merely input variables. The defining condition is also part of the post-condition but for defining output variables in terms of their relation with input variables.

For example, let the process Test have the following specification:

```

process Test(x: bool, y: real ) z: string
  ext rd w: bool
  pre w = true
  post x = true and y < 0 and z = "output_1" or
        x = true and y = 0 and z = "output_2" or
        x = true and y > 0 and z = "output_3" or
        x = false and y < 0 and z = "output_4" or
        x = false and y = 0 and z = "output_5" or
        x = false and y > 0 and z = "output_6"
  end_process

```

In this specification, the post-condition is given as a disjunction of six functional scenarios. Each functional scenario is a conjunction of a guard condition and the corresponding defining condition, omitting the pre-condition of the process in the original concept of functional scenario, because it applies to every such a conjunction in the post-condition. For example,  $x = \text{true}$  and  $y < 0$  and  $z = \text{"output\_1"}$  is a functional scenario whose guard condition is  $x = \text{true}$  and  $y < 0$  and defining condition is  $z = \text{"output\_1"}$ . The original functional scenario should be  $w = \text{true}$  and  $x = \text{true}$  and  $y < 0$  and  $z = \text{"output\_1"}$ .

## 2.3 Scenario-Based Formal Specification

By scenario-based formal specification, we mean that the post-condition of a process is written by gradually adding functional scenarios one by one [6]. For instance, the specification of the above process Test can be completed by gradually adding more functional scenarios, as illustrated in Table 1.

This table shows a simple idea of gradually enriching the post-condition of the process specification. Initially, the first functional scenario is written in the post-condition by the developer, which also forms the current version of the post-condition. Then, the second functional scenario is written to extend the existing post-condition to form a new current version. This process continues until all of the necessary functional scenarios are added in turn to form the final version of the post-condition.

In such a process of completing the post-condition, there are two concerns. One is whether each added functional scenario is internally consistent or not. For example, is

the first functional scenario in the example above,  $x = true$  and  $y < 0$  and  $z = "output\_1"$ , satisfiable? If the answer is yes, the scenario is said to be consistent; otherwise, it is not consistent. Obviously, this scenario is consistent. But it will become inconsistent if it is changed to  $x = true$  and  $y < 0$  and  $y < 0$  and  $z = "output\_1"$ . Another concern is whether the post-condition is complete or not. By completeness we mean that for any input satisfying the pre-condition, there must exist an output that satisfies the post-condition, that is, one of the functional scenario of the post-condition. Our strategy for predicating further contents of the specification from the current partial version is to achieve the completeness of the specification.

**Table 1.** Scenario-based formal specification.

No.	New functional scenario added to the post-condition	Current version of the post-condition
1	$x = true$ and $y < 0$ and $z = "output\_1"$	$x = true$ and $y < 0$ and $z = "output\_1"$
2	$x = true$ and $y = 0$ and $z = "output\_2"$	$x = true$ and $y < 0$ and $z = "output\_1"$ or $x = true$ and $y = 0$ and $z = "output\_2"$
3	$x = true$ and $y > 0$ and $z = "output\_3"$	$x = true$ and $y < 0$ and $z = "output\_1"$ or $x = true$ and $y = 0$ and $z = "output\_2"$ or $x = true$ and $y > 0$ and $z = "output\_3"$
4	$x = false$ and $y < 0$ and $z = "output\_4"$	$x = true$ and $y < 0$ and $z = "output\_1"$ or $x = true$ and $y = 0$ and $z = "output\_2"$ or $x = true$ and $y > 0$ and $z = "output\_3"$ or $x = false$ and $y < 0$ and $z = "output\_4"$
...	...	...

### 3 Principles

In this section, we discuss the basic principle for checking internal consistency, predicating contents of specifications, and forming the tabular form in order to prevent errors related to specification consistency and completeness and to improve the specification readability. The assumption for our work in this paper is that the post-condition of the process under consideration must be written gradually in the scenario-based manner. Of course, there are other styles of writing formal specifications, but since each style requires a different strategy for predication, we only focus on the scenario-based style that has proved to be the most effective in practice in accordance with the second author's experience over the last twenty-five years.

#### 3.1 Checking Internal Consistency

Checking the internal consistency is done at the functional scenario level. Whenever a new functional scenario is added to the current version of the post-condition of the process specification, its internal consistency will be automatically checked. Checking

the syntax and type consistency does not fall into our scope because they can be checked by a compiler (or similar). What we are focusing on here is to check the semantic consistency in order to ensure that the functional scenario is satisfiable.

Let  $f$  denote a functional scenario  $G$  and  $D$  ( $G$  is the guard condition and  $D$  the defining condition) written by the developer (the user of our tool) with the following structure of the guard condition  $G$ :

$$G_1 \text{ and } G_2 \text{ and } \dots \text{ and } G_n.$$

The consistency checking is done through automatic testing. That is, a set of test data is generated to evaluate whether the guard condition can be evaluated to *true*. If yes, the checking will terminate; otherwise, more test data will be produced until either the guard condition proves to be satisfiable or a predefined number of failure to do so has been reached.

Several factors may affect the effectiveness or efficiency of the checking. One is how the test data are generated. As far as this point is concerned, we adopt the techniques for automatic test data generation from predicates developed by Liu and NaKajima in [7]. Another factor is how to determine the number of test data that do not evaluate the guard condition to *true* as the criterion to terminate the testing and to support the conclusion that the guard condition is not consistent. This challenge has not been successfully resolved; what we can do is to let the developer decide based on his or her engineering judgement.

### 3.2 Automatic Checking the Internal Consistency

Let  $f$  denote the first functional scenario  $G$  and  $D$  ( $G$  is the guard condition and  $D$  the defining condition) written by the developer (the user of our tool) with the following structure of the guard condition  $G$ :

$$G_1 \text{ and } G_2 \text{ and } \dots \text{ and } G_n.$$

Where each  $G_i$  ( $i = 1, \dots, n$ ) is a relation (atomic predicate). Then, the following specification content will be automatically predicated in principle:

$$\begin{aligned} &G_1 \text{ and } G_2 \text{ and } \dots \text{ and not } G_n \text{ and } D_1 \text{ or} \\ &G_1 \text{ and } G_2 \text{ and } \dots \text{ and not } G_{n-1} \text{ and not } G_n \text{ and } D_2 \text{ or} \\ &\dots \\ &\text{not } G_1 \text{ and not } G_2 \text{ and } \dots \text{ and not } G_n \text{ and } D_m. \end{aligned}$$

In the above expression,  $m = 2^n - 1$ . The predicated content should take both *true* and *false* case for each constituent predicate  $G_i$  into account, where the total number of the predicated functional scenarios is  $2^n - 1$ , the corresponding defining condition  $D_j$  ( $j = 1, \dots, 2^n - 1$ ) may be the same for some different guard conditions. Therefore, in practice, many of the predicated functional scenarios can be merged and therefore the



less number of functional scenarios will be ultimately formed for the process specification. The important property of such a specification is that its completeness, as explained previously, is assured in the final specification.

When forming *not*  $G_i$  for each relation  $G_i$ , we also take different methods for different kinds of relations. Specifically, we divide a relation into three kinds:

- (1) Comparison between a string type variable and a string constant,
- (2) Comparison between a boolean variable and a truth value, and
- (3) Comparison between numeric variables and values.

For case (1), let us consider the example  $x = \text{“Hosei University”}$ . In this case, the predicated negation (i.e., *not*  $G_i$ , where  $G_i$  denotes this equality) is:

$$x \neq \text{“Hosei University”}$$

Which means that  $x$  is not equal to *“Hosei University”*. For case (2), let us consider the example  $y = \text{true}$ . In this case, the predicated negation is:

$$y = \text{false.}$$

For case (3), let us consider the example  $y > 3$ . In this case, the predicated negation is:

$$y < 3 \text{ or } y = 3.$$

Note that our algorithms for predication deal with the expressions generally, and the reason we only use examples here for the discussion is to ensure a good readability of the presentation so that the reader can easily understand our essential idea.

### 3.3 Automatic Translation to Tabular Form

The construction of tabular forms from the SOFL process specification is functional scenario-based. Each scenario is translated into one tabular form, and the whole functional scenario form of the specification, which is a disjunction of all functional scenarios, is translated into a set of related tabular forms. Table 2 is an example of the tabular form of one scenario. The table mainly includes two parts: *guard condition part* and *defining condition part*, which are represented by *GC* and *DC* as well as the related contents, respectively. The content of *GC* is further divided into a number of rows, each

**Table 2.** The tabular form of scenario.

<i>Functional scenario</i>		
<i>GC</i>	$x = \text{true}$	✓
	$y = 0$	✓
<i>DC</i>	$z = 1$	✓

showing a relation (atomic constitute predicate) of the guard condition. If each relation is marked with a tick symbol to draw the attention from the specification reader. The same presentation style also applies to *DC* part.

To build a tabular form for a process specification, the following three steps are taken:

**Step1:**

- (1) Get all of the related information of variables.
- (2) Obtain all of the functional scenarios from the specification.

**Step2:**

- (1) Divide scenarios and predicates based on keywords, such as “and”, “or”.
- (2) Find the guard condition or defining condition from each functional scenario.

**Step3:**

- (1) Draw tabular forms based on scenarios.
  - (2) Show tabular forms to the user, and provide explanative messages to the user.
- These three steps have properly implemented into our supporting tool.

## 4 Software Tool

We have developed a prototype tool to support the scenario-based formal specification approach [8–12]. In this section, we will focus our discussion on the structure and functions of the tool, and also present some case studies to demonstrate the feasibility and usefulness.

### 4.1 Implementation Structure

The techniques for realizing the important functions have been implemented to certain extent in our tool. The name of this tool is *Scenario-based SOFL Writing Supported Tool*. Figure 2 shows the main GUI of the tool. The tool also includes conventional functions, such as *open file*, *save file*, and *edit text*. The GUI mainly has three areas, the management area (left part), the operating area (middle part), and the feedback area (right part). The management area can show the structure of a module and the content of a module. When the user selects a process in a SOFL module, its specification will be displayed in the operating area. The user can edit the specification as he or she wishes. In the feedback area, the user can select pieces of information displayed, such as the set of scenarios and the feedback message of the process. Tabular forms produced by translation can also be displayed in this area.

### 4.2 Case Study

We use the tool to construct a process specification to test the usefulness of the three functions supported by the tool. Since it is rather difficult to find a realistic operation suitable for exploring various functions mentioned previously, we use an artificial example suitable for our purpose in the case study. The operation Test is given as a process in SOFL below.

```

process Test( var1: string, var2: bool ) var4: string
ext rd var3: real
pre true
post .....
end_process

```

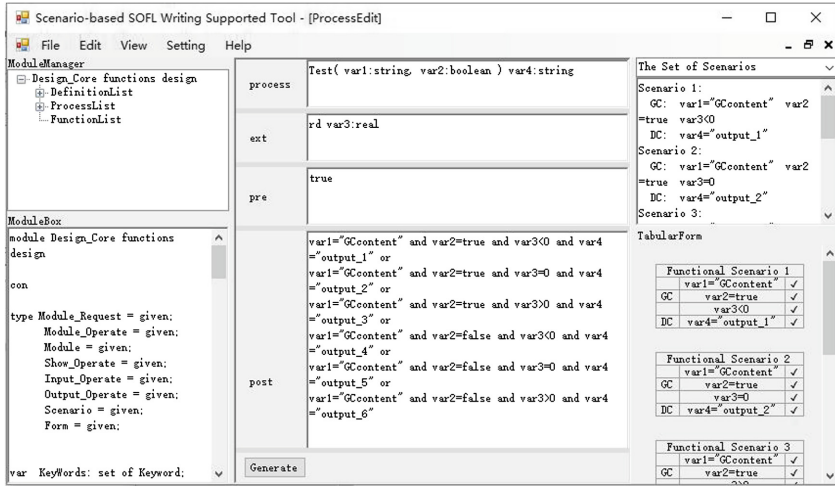


Fig. 2. The UI of the tool.

The process includes two input variables  $var1$  and  $var2$ , one output variable  $var4$ , and one external (or state) variable  $var3$ . Before the user writes the post-condition, the tool first analyzes the process signature to get the necessary information of all the variables and stores them in a table as shown in Table 3.

Table 3. Variables.

Variable name	Variable type	Condition type
var1	string	Guard condition
var2	bool	Guard condition
var3	real	Guard condition
var4	string	Defining condition

On the basis of the information in this table, the tool will automatically predicate two functional scenarios as shown in Table 4, one describing the situation when the Boolean variable  $var2$  is true and the other defining the situation when it is false.

Responding to the output of the tool, the user will then write specification contents for the conditions  $C(var1)$ ,  $C(var3)$ , and  $C(var4)$  in both functional scenarios. If the user makes logical errors in writing the conditions, the tool is supposed to

**Table 4.** The first functional scenarios predicated by the tool.

Existing information			The set of scenarios
Item	Name	Type	Scenario 1:
Input variable	var1	string	GC: $C(var1)$ , $var2 = true$ , $C(var3)$
	var2	bool	DC: $C(var4)$
External variable	var3	real	Scenario 2:
Output variable	var4	string	GC: $C(var1)$ , $var2 = false$ , $C(var3)$
			DC: $C(var4)$

automatically check it and provide a feedback message. There are three possible logical errors as shown in Table 5. The first error is when  $var1 = "GCcontent"$  is false, the tool will update the set of scenarios. It only adds a new scenario that includes the failure of  $var1 = "GCcontent"$ .

**Table 5.** Functional scenarios with logical errors.

Item		The set of scenarios
Pre-condition	True	Scenario 1: GC: $var1 = "GCcontent"$ , $var2 = true$ , $C(var3)$ DC: $C(var4)$
Post-condition	not $var1 = "GCcontent"$ and $var2 = true$	Scenario 2: GC: not $var1 = "GCcontent"$ , $var2 = false$ , $C(var3)$ DC: $C(var4)$ Scenario 3: GC: $var1 = "GCcontent"$ , $var2 = false$ , $C(var3)$ DC: $C(var4)$

The second one is the different value range in predicates. We do not specify the value of  $var3$  in the beginning. So when the user adds a condition about  $var3$  and set the value of  $var3$ , it means a new value has been generated, and a mismatching appears. So we need to modify the set of scenarios. The new set of scenarios is shown in Table 6.

The third error is the self-contradiction of the defining condition. The tool will not build a new scenario in the set of scenarios. Because the defining condition cannot affect the structure of the set of scenario. It only represents the result of a scenario. So, we only need to check the correction, then modify it.

Finally, a fallible situation will be shown to the user: different scenarios have the same input and the different output. We can see, if  $var2$  is equal to true, and  $var3$  is greater than 0, this situation satisfies both Scenario 1 and Scenario 2, so the defining condition might be  $var4 = "output_1"$  or  $var4 = "output_2"$ . In SOFL, this situation is allowed. For error prevention, we can show this feedback message to the user, as shown in Table 7.

**Table 6.** The change of value range.

<i>Post-condition</i>	<i>The set of scenarios</i>
<i>var1 = "GCcontent" and var2 = true and var3 &lt;=0</i>	<p><i>Scenario 1:</i> GC: <i>var1 = "GCcontent", var2 = true, var3 &lt;=0</i> DC: <i>C(var4)</i></p> <p><i>Scenario 2:</i> GC: <i>var1 = "GCcontent", var2 = true, var3 &gt; 0</i> DC: <i>C(var4)</i></p> <p><i>Scenario 3:</i> GC: <i>var1 = "GCcontent", var2 = false, var3 &lt;=0</i> DC: <i>C(var4)</i></p> <p><i>Scenario 4:</i> GC: <i>var1 = "GCcontent", var2 = false, var3 &gt; 0</i> DC: <i>C(var4)</i></p>

**Table 7.** The confliction between different scenarios.

<i>Post-condition</i>	<i>The set of scenarios</i>	<i>Feedback message</i>
<i>var1 = "GCcontent" and var2 = true and var3 &gt;=0 and var4 = "output_1" or var1 = "GCcontent" and var2 = true and var3 &gt; 0 and var4 = "output_2"</i>	<p><i>Scenario 1:</i> GC: <i>var1 = "GCcontent", var2 = true, var3 &gt;=0</i> DC: <i>var4 = "output_1"</i></p> <p><i>Scenario 2:</i> GC: <i>var1 = "GCcontent", var2 = true, var3 &gt; 0</i> DC: <i>var4 = "output_2"</i></p> <p>.....</p>	<i>A contradiction between scenarios. Scenario 1 and Scenario 2.</i>

<table border="1"> <thead> <tr> <th colspan="3">Functional Scenario 1</th> </tr> </thead> <tbody> <tr> <td></td> <td>var1="GCcontent"</td> <td>✓</td> </tr> <tr> <td>GC</td> <td>var2=true</td> <td>✓</td> </tr> <tr> <td></td> <td>var3&lt;0</td> <td>✓</td> </tr> <tr> <td>DC</td> <td>var4="output_1"</td> <td>✓</td> </tr> </tbody> </table>	Functional Scenario 1				var1="GCcontent"	✓	GC	var2=true	✓		var3<0	✓	DC	var4="output_1"	✓	<table border="1"> <thead> <tr> <th colspan="3">Functional Scenario 4</th> </tr> </thead> <tbody> <tr> <td></td> <td>var1="GCcontent"</td> <td>✓</td> </tr> <tr> <td>GC</td> <td>var2=false</td> <td>✓</td> </tr> <tr> <td></td> <td>var3&lt;0</td> <td>✓</td> </tr> <tr> <td>DC</td> <td>var4="output_4"</td> <td>✓</td> </tr> </tbody> </table>	Functional Scenario 4				var1="GCcontent"	✓	GC	var2=false	✓		var3<0	✓	DC	var4="output_4"	✓
Functional Scenario 1																															
	var1="GCcontent"	✓																													
GC	var2=true	✓																													
	var3<0	✓																													
DC	var4="output_1"	✓																													
Functional Scenario 4																															
	var1="GCcontent"	✓																													
GC	var2=false	✓																													
	var3<0	✓																													
DC	var4="output_4"	✓																													
<table border="1"> <thead> <tr> <th colspan="3">Functional Scenario 2</th> </tr> </thead> <tbody> <tr> <td></td> <td>var1="GCcontent"</td> <td>✓</td> </tr> <tr> <td>GC</td> <td>var2=true</td> <td>✓</td> </tr> <tr> <td></td> <td>var3=0</td> <td>✓</td> </tr> <tr> <td>DC</td> <td>var4="output_2"</td> <td>✓</td> </tr> </tbody> </table>	Functional Scenario 2				var1="GCcontent"	✓	GC	var2=true	✓		var3=0	✓	DC	var4="output_2"	✓	<table border="1"> <thead> <tr> <th colspan="3">Functional Scenario 5</th> </tr> </thead> <tbody> <tr> <td></td> <td>var1="GCcontent"</td> <td>✓</td> </tr> <tr> <td>GC</td> <td>var2=false</td> <td>✓</td> </tr> <tr> <td></td> <td>var3=0</td> <td>✓</td> </tr> <tr> <td>DC</td> <td>var4="output_5"</td> <td>✓</td> </tr> </tbody> </table>	Functional Scenario 5				var1="GCcontent"	✓	GC	var2=false	✓		var3=0	✓	DC	var4="output_5"	✓
Functional Scenario 2																															
	var1="GCcontent"	✓																													
GC	var2=true	✓																													
	var3=0	✓																													
DC	var4="output_2"	✓																													
Functional Scenario 5																															
	var1="GCcontent"	✓																													
GC	var2=false	✓																													
	var3=0	✓																													
DC	var4="output_5"	✓																													
<table border="1"> <thead> <tr> <th colspan="3">Functional Scenario 3</th> </tr> </thead> <tbody> <tr> <td></td> <td>var1="GCcontent"</td> <td>✓</td> </tr> <tr> <td>GC</td> <td>var2=true</td> <td>✓</td> </tr> <tr> <td></td> <td>var3&gt;0</td> <td>✓</td> </tr> <tr> <td>DC</td> <td>var4="output_3"</td> <td>✓</td> </tr> </tbody> </table>	Functional Scenario 3				var1="GCcontent"	✓	GC	var2=true	✓		var3>0	✓	DC	var4="output_3"	✓	<table border="1"> <thead> <tr> <th colspan="3">Functional Scenario 6</th> </tr> </thead> <tbody> <tr> <td></td> <td>var1="GCcontent"</td> <td>✓</td> </tr> <tr> <td>GC</td> <td>var2=false</td> <td>✓</td> </tr> <tr> <td></td> <td>var3&gt;0</td> <td>✓</td> </tr> <tr> <td>DC</td> <td>var4="output_6"</td> <td>✓</td> </tr> </tbody> </table>	Functional Scenario 6				var1="GCcontent"	✓	GC	var2=false	✓		var3>0	✓	DC	var4="output_6"	✓
Functional Scenario 3																															
	var1="GCcontent"	✓																													
GC	var2=true	✓																													
	var3>0	✓																													
DC	var4="output_3"	✓																													
Functional Scenario 6																															
	var1="GCcontent"	✓																													
GC	var2=false	✓																													
	var3>0	✓																													
DC	var4="output_6"	✓																													

**Fig. 3.** The tabular forms.

After all of the logical errors are removed, the tool will automatically generate the tabular forms in the real-time manner. When a scenario has been written, the tool will show all tabular forms of the scenario. Figure 3 shows the final result of the generation.

## 5 Related Work

There are some studies about the development or analysis of SOFL. Zhu and Liu have presented a way to analyze SOFL [13], and have built a tool to support it. One of fundamental functions is very important in this tool: Syntactic analysis of SOFL formal specifications and automatic generation of functional scenarios. Another paper is written by Liu, Hayashi, Takahashi, Kimura and Nakayama, they first explain the concepts of the functional scenario form and the functional scenario in the context of a VDM operation specification, then discuss the techniques for the transformation [14]. For capturing Complete and Accurate Requirement, Liu gives a series of theories and explains how it can be achieved [4]. The readability of specifications for large-scale and complex systems can be so poor, Liu and Wang describe a software tool that supports the animation of specifications by simulating their functional scenarios using the Message Sequence Chart (MSC) to review the adequacy of the specification [15].

For this paper, the most important related research is syntactic analysis of SOFL. These related works provide some effective methods can be used to help us get the most basic expression of post-condition. We can also further develop SOFL in completeness, requirement, verification and convenience.

## 6 Conclusion and Future Work

We have developed a prototype tool to support the functional scenario-based formal specification approach that proves to be effective in practice. The tool offers three functions, including automatic checking of the internal consistency of functional scenarios, automatically predicating more functional scenarios based on the current ones, and automatically translating the formal specification into tabular forms to improve its readability. We have conducted a case study that helps us demonstrate the usefulness of the tool in preventing errors during the construction of the formal specifications.

In the future, we are interested in continuous working on the extension and improvement of the tool to support more functions in a robust manner, including automatic checking of internal consistency within functional scenarios and between functional scenarios of different processes. Using the updated tool, we will also be interested in conducting larger experiments to evaluate both the scenario-based formal specification approach and the tool support.

**Acknowledgments.** This work was supported by JSPS KAKENHI Grant Number 26240008

## References

1. Liu, S., Chen, Y., Nagoya, F., McDermid, J.: Formal specification-based inspection for verification of programs. *IEEE Trans. Softw. Eng.* **35**(8), 1100–1122 (2012)
2. Luo, J., Liu, S., Wang, Y., Zhou, T.: Applying SOFL to a railway interlocking system in industry. In: Liu, S., Duan, Z., Tian, C., Nagoya, F. (eds.) *SOFL + MSVL 2016*. LNCS, vol. 10189, pp. 160–177. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-57708-1\\_10](https://doi.org/10.1007/978-3-319-57708-1_10)
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. Liu, S.: Capturing complete and accurate requirements by refinement. In: *Proceedings of 8th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society Press, Greenbelt, 2–4 December (2002)
5. Liu, S.: *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer, Heidelberg (2004)
6. Li, M., Liu, S.: Automated functional scenarios-based formal specification animation. In: *Software Engineering Conference (APSEC)*, Hong Kong, China (2012)
7. Liu, S., Nakajima, S.: A decompositional approach to automatic test case generation based on formal specifications. In: *Secure Software Integration and Reliability Improvement*, Singapore (2010)
8. Martin, S., Rudy, S., Josep, H.: Enhanced latent semantic analysis by considering mistyped words in automated essay scoring. In: *Informatics and Computing (ICIC)*, Mataram, Indonesia (2017)
9. Sahand, S., Mehrnaz, A., Bijan, A.: Systematic approximate logic optimization using don't care conditions. In: *Quality Electronic Design (ISQED)*, Santa Clara, CA, USA (2017)
10. Senem, K., Bahar, K., Tank, K.: Attribute value-range detection in identification of paraphrase sentence pairs. In: *Signal Processing and Communication Application Conference (SIU)*. Zonguldak, Turkey (2016)
11. Luo, X., Liu, S., Wu, H.: A framework for transforming SOFL formal specifications to programs. In: *Software Engineering and Service Science (ICSESS)*, Beijing, China (2015)
12. Chen, H., Shen, Y., Jiang, J.: Extended SOFL features for the modeling of middleware-based transaction management. In: *Engineering of Complex Computer Systems (ICECCS)*, Shanghai, China (2005)
13. Zhu, S., Liu, S.: A supporting tool for syntactic analysis of sofl formal specifications and automatic generation of functional scenarios. In: Liu, S., Duan, Z. (eds.) *SOFL + MSVL 2013*. LNCS, vol. 8332, pp. 104–117. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-04915-1\\_8](https://doi.org/10.1007/978-3-319-04915-1_8)
14. Liu, S., Hayashi, T., Takahashi, K., Kimura, K., Nakayama, T., Nakajima, S.: Automatic transformation from formal specifications to functional scenario forms for automatic test case generation. In: *9th International Conference on Software Methodologies, Tools, and Techniques*. IOS Press, Yokohama (2010)
15. Liu, S., Wang, H.: An automated approach to specification animation for validation. *J. Syst. Softw.* **80**, 1271–1285 (2007)



# A Proof Score Approach to Formal Verification of an Imperative Programming Language Compiler

Dorian Daudier<sup>1</sup>, Trinh Ngoc Quoc Bao<sup>2</sup>, and Kazuhiro Ogata<sup>3</sup>

<sup>1</sup> ISAE-ENSMA, Chasseneuil-du-Poitou, France  
dorian.daudier@etu.isae-ensma.fr

<sup>2</sup> VienthongA, Ho Chi Minh City, Vietnam  
quocbao.tn@gmail.com

<sup>3</sup> JAIST, Nomi, Japan  
ogata@jaist.ac.jp

**Abstract.** An interpreter for an imperative programming language called Minila has been formally specified in CafeOBJ, an executable specification language, and so have a virtual machine (VM) and a compiler. The compiler transforms a Minila program into an instruction sequence processed by the VM. Since the formal specifications are executable, it is doable to test if for any concrete terminating program  $p$  the result of interpreting  $p$  is the same as the one of processing by the VM the instruction sequence generated from  $p$  by the compiler, where the result is an environment, a variable-value pair collection. The equivalence is called the Minila compiler correctness with respect to  $p$ . In addition to test, properties of CafeOBJ specifications can be theorem proved by writing what are called proof scores in CafeOBJ and executing them with CafeOBJ. The Minila compiler correctness for all terminating programs in Minila has been theorem proved.

**Keywords:** Algebraic specification · CafeOBJ · Compiler  
Formal verification · Proof score · Theorem proving

## 1 Introduction

Human beings have heavily rely on software. It is impossible to even think about our life without software. We have been facing what is called Internet of Things (IoT) era in which almost all things are connected through the Internet and software will become more and more important. Software must be trustworthy in

---

The research was first conducted by the second author (A2) [1] partially supervised by the third author (A3) when A2 was a Master's student of JAIST and then reconducted by the first author (A1) [2] supervised by A3 totally independent from A2 when A1 was an internship student of JAIST. The paper is based on the first author's achievement. This work was partially supported by JSPS KAKENHI Grant Number 26240008.



the era. Main part of software is developed in high-level programming languages from which low-level instruction sequences that can be dealt with by microprocessors or virtual machines are generated with compilers that are also software. Therefore, trustworthiness of software crucially depends on compilers. Almost all compilers, however, have been certified with testing only and therefore safety-critical project teams as well as compiler suppliers need to carefully check if the compilers used in the projects conform to the semantics of the programming language used [3]. Although testing is one important technique to certify software, engineers as well as researchers in software have realized that it does not suffice to truly certify software with testing only. Formal verification is one possible promising technique that is complementary to testing to truly certify software including compilers.

An interpreter for an imperative programming language called Minila<sup>1</sup> has been formally specified in CafeOBJ [4] ([cafeobj.org](http://cafeobj.org)), an executable specification language, and so have a virtually machine (VM) and a compiler. The compiler transforms a Minila program into an instruction sequence processed by the VM. A CafeOBJ specification consists of a signature  $\Sigma$  and a set  $E$  of equations<sup>2</sup>.  $\Sigma$  consists of a set of sorts including partial-order among sorts and a set of operators. Sorts are interpreted as sets of data values, partial-order among sorts are then subset relations among those sets, and operators are basically interpreted as functions over those sets, although some operators called constructors are used to represent data values. Equations define properties of functions and data values. An operator is in the form  $f : S_1 \dots S_n \rightarrow S_{n+1}$ , where  $n \geq 0$ ,  $f$  is the operator name, each  $S_i$  is a sort, the sequence of sorts  $S_1 \dots S_n$  is an arity of  $f$ , and  $S_{n+1}$  is the sort of  $f$ . If  $n = 0$ , the operator is called a constant. An operator may be a constructor. If that is the case, `constr` follows the sort of the operator. Terms are made of operators and variables. A variable of a sort is a term of the sort. Given terms  $t_1, \dots, t_n$  of sorts  $S_1, \dots, S_n$ ,  $f(t_1, \dots, t_n)$  is a term of sort  $S_{n+1}$ . If a sort  $S'$  is a super-sort of a sort  $S$  (and then  $S$  is a sub-sort of  $S'$ ), terms of  $S$  are also those of  $S'$ . If the operator name contains underscores `_`, parameters are put at those underscores. For example, for `if_then_else_fi : Bool S S -> S`, `if b then t1 else t2 fi` is used as a term of  $S$ , where `Bool` is the sort of built-in Boolean values,  $b$  is a term of `Bool`, and  $t_1, t_2$  are terms of  $S$ . An equation is in the form  $l = r$ , where  $l, r$  are terms of a same sort. An equation can have a condition and is in the form  $l = r$  `if c`, where  $c$  is the condition and a term of `Bool`. An equation can be used as a left-to-right rewrite rule. If it has a condition, it can be used as a left-to-right rewrite rule when the condition reduces to `true` that is the constant of `Bool` and is given the expected meaning. This is how CafeOBJ specifications can be executed.

<sup>1</sup> Minila consists of a minimal essence of procedural programming languages, such as C, and has been used by the third author, et al. for educational purposes ([www.jaist.ac.jp/~ogata/lecture/i217/](http://www.jaist.ac.jp/~ogata/lecture/i217/)).

<sup>2</sup> A set of trans rules could be included in a CafeOBJ specification, but trans rules are not used in this paper.

Since CafeOBJ specifications can be executed, it is doable to test if for any concrete terminating Minila program  $p$  the result of interpreting  $p$  is the same as the one of processing by the VM the instruction sequence generated from  $p$  by the compiler, where the result is an environment that is a variable-value pair collection and used to record the value of each Minila variable in  $p$ . The equivalence is called the Minila compiler correctness with respect to  $p$ .

In addition to testing, properties of CafeOBJ specifications can be theorem proved by writing what are called proof scores in CafeOBJ and executing them with CafeOBJ. The Minila compiler correctness for all terminating Minila programs has been theorem proved. The paper describes the theorem proof of the Minila compiler correctness for all terminating Minila programs.

It is one strong point of CafeOBJ's that CafeOBJ specifications can be used for both theorem proving and testing thanks to their executability. Formal specifications written in some formal specification languages, such as VDM++, can be tested because a class of VDM++ specifications can be executed, but cannot be theorem proved. Formal specifications written in a language dedicated to theorem proving, such as Isabelle/HOL and Coq, cannot be executed and then cannot be tested, while executable programs can be derived from proofs conducted by Coq, which is one strong point of Coq's.

The rest of the paper is organized as follows. Section 2 briefly introduces formal specification and verification in CafeOBJ. Section 3 describes Minila. Formal verification of the Minila compiler is described in Sect. 4. Section 5 discusses some issues related to the formal verification of the Minila compiler. Finally the paper is concluded in Sect. 6.

## 2 Formal Specification and Verification in CafeOBJ

CafeOBJ is an algebraic specification language. Main ingredients of CafeOBJ specifications are equations. As described, CafeOBJ specifications can be executed by using equations as left-to-right rewrite rules with the CafeOBJ system. Thanks to this executability, it is possible to not only test CafeOBJ specifications like ordinary programs, but also theorem prove properties of CafeOBJ specifications by writing what are called proof scores<sup>3</sup> in CafeOBJ and execute them with the CafeOBJ system.

Let us consider the specification of generic lists:

```
mod! LIST (E :: TRIV) {
  pr(PNAT)
  [List]
  op nil : -> List {constr}
  op |_| : Elt.E List -> List {constr} .
  op @_ : List List -> List .
```

---

<sup>3</sup> The terminology “proof score” has been coined by Joseph Goguen [5]. The authors suppose that since he was very enthusiastic to music, the terminology came from “music score”.

```

op len : List -> PNat .
op rev1 : List -> List .
op rev2 : List -> List .
op sr2 : List List -> List .
vars E E2 : Elt.E .
vars L1 L2 L3 : List .
eq [01]: nil @ L2 = L2 .
eq [02]: (E | L1) @ L2 = E | (L1 @ L2) .
eq len(nil) = 0 .
eq len(E | L1) = s(len(L1)) .
eq rev1(nil) = nil .
eq rev1(E | L1) = rev1(L1) @ (E | nil) .
eq rev2(L1) = sr2(L1,nil) .
eq sr2(nil,L2) = L2 .
eq sr2(E | L1,L2) = sr2(L1,E | L2) .
}

```

Basic units of CafeOBJ specifications are modules. `PNAT` is a module in which Peano natural numbers are specified and reused in the module `LIST` that has a parameter `E`. `TRIV` is a built-in module in which one sort `Elt` is declared. An actual parameter `LIST` can take is basically a module in which there must be at least one sort. `Elt.E` is the sort `Elt` in the parameter `E` and used as the sort of list elements. `List` is the sort of lists. `nil` and `_|_` are the constructors of lists. `nil` is the empty list and given elements `a`, `b` and `c` of the sort `Elt.E`, `a | b | c | nil` is the list that consists of the three elements in this order. `PNat` is the sort for Peano natural numbers and `0` and `s` are the constructors of Peano natural numbers, representing zero and the successor function. `_@_`, `len`, `rev1`, `rev2` and `sr2` are functions that are defined in terms of equations. Equations can be given labels, such as `@1` and `@2`. `E` and `E2` are variables of the sort `Elt.E`, and `L1`, `L2` and `L3` are variables of the sort `List`. Equations are used as left-to-right rewrite rules to rewrite terms. For example, `(a | b | nil) @ (c | d | nil)` is rewritten as follows:

```

(a | b | nil) @ (c | d | nil)
→@2 a | ((b | nil) @ (c | d | nil))
→@2 a | b | (nil @ (c | d | nil))
→@1 a | b | c | d | nil

```

where  $\rightarrow_{@1}$  and  $\rightarrow_{@2}$  mean one-step rewrites with the equations `@1` and `@2`, respectively.

`_@_` concatenates two lists. `len` returns the number of elements in a list. `rev1` reverses a list, and so does `rev2`. `rev1` is defined in the very standard way, while `rev2` is defined in a tail recursive call and uses `sr2`. `rev2` may reverse a list faster than `rev1`, but does `rev2` really reverse any list? One possible way to confirm if it does is to test several possible cases:

```

open LIST .
ops a b c d : -> Elt.E .

```

```

red rev1(nil) = rev2(nil) .
red rev1(a | nil) = rev2(a | nil) .
red rev1(a | b | nil) = rev2(a | b | nil) .
red rev1(a | b | c | nil) = rev2(a | b | c | nil) .
red rev1(a | b | c | d | nil) = rev2(a | b | c | d | nil) .
close

```

where `rev1` is used as an oracle. `open` begins the use of a module and `close` stops the use of the module. An open-close environment makes an on-the-fly module in which a given module is imported. Therefore, what can be declared in modules can be declared in open-close environments, such as constants. Command `red` reduces a given term. Each of the five reductions in the open-close environment returns `true`, meaning that `rev2` really reverses any list that consists of up to four elements. To the best of our knowledge, parametrized modules, such as functors in Standard ML and generics in Java, must be instantiated to use them. CafeOBJ parametrized modules can be, however, used as they are without instantiating them. Thus, the four constants of `Elt.E` denote really arbitrary elements, and then the five reductions guarantee that `rev2` reverses an arbitrary list that consists of up to four elements.

The test does not, however, guarantee that `rev2` really reverses a list that consists of five or more elements. One possible way to guarantee if it really reverses every list is to prove that `rev2(L) = rev1(L)` for all `L>List`. The proof can be carried out by structural induction on `L`. The proof score is as follows:

```

open LIST .
red rev1(nil) = rev2(nil) .
close
open LIST .
op l1 : -> List . op e : -> Elt.E .
eq [IH]: rev1(l1) = rev2(l1) .
eq [lem1]: sr2(L1,E2 | L2) = sr2(L1,nil) @ (E2 | L2) .
red rev1(e | l1) = rev2(e | l1) .
close

```

The first open-close fragment is the base case and the second one is the induction case. `l1` and `e` are fresh constants that represent an arbitrary list and an arbitrary element. The equation `IH` is the induction hypothesis and the equation `lem1` is a lemma. Both reductions return `true` meaning that the proof succeeds, provided that the lemma is proved. The proof of the lemma needs other lemmas:

```

eq [lem2]: (L1 @ L2) @ L3 = L1 @ (L2 @ L3) .
eq [lem3]: rev1(rev1(L1) @ L2) = rev1(L2) @ L1 .
eq [lem4]: L1 @ nil = L1 .

```

If `lem1` is not used, the second reduction returns

```

sr2(l1,nil) @ (e | nil) = sr2(l1,e | nil)

```

This allows us to conjecture one lemma candidate:

```
eq [lem0]: sr2(L1,nil) @ (E2 | nil) = sr2(L1,E2 | nil) .
```

However, the proof of `lem0` requires a series of similar lemmas because `nil` and `E2 | nil` in `lem0` are too specific. One possible remedy is to make them more generic, obtaining `lem1`. The reason why we do not use

```
eq [lem1']: sr2(L1,nil) @ (E2 | L2) = sr2(L1,E2 | L2) .
```

is that CafeOBJ uses equations as left-to-right rewrite rules. There is no essential difference between `lem1` and `lem1'` from an equational reasoning point of view because there is no direction, while there is a significant difference between them from a rewriting point of view because there is the left-to-right direction.

### 3 Minila

Minila is an imperative programming language and has five kinds of statements: the empty statement (`estm`), assignment statements ( $V := E$  ;), conditional statements (`if E {S} else {S}`), loop statements (`while E {S}`) and sequential composition statements ( $S S$ ), where  $S$  is a statement and  $E$  is an expression. Natural numbers are only the data type available in Minila. The following is a program (`Psr`) in Minila that computes the positive integral part of the square root of the natural number stored in the variable `x`:

```
x := 20000000000000000 ;
y := 0 ;
z := x ;
while y != z {
  if ((z - y) % 2) == 0 {
    tmp := y + (z - y) / 2 ;
  } else {
    tmp := y + ((z - y) / 2) + 1 ;
  }
  if tmp * tmp > x {
    z := tmp - 1 ;
  } else {
    y := tmp ;
  }
}
```

Note that Peano natural numbers are used in the specifications that have been used in the formal verification. As usual, non-zero is treated as true, while zero is treated as false. All operators except for some used in Minila are very common. The exceptions are `!=`, `==` and `-`. For the first two, the notations are different because `=` and `==` are given specific meanings in CafeOBJ, and for the third one, it is given a different semantics because natural numbers are only the data type used in Minila. Given two natural numbers `m` and `n`, `m != n` is 0 if `m` equals `n`

and 1 otherwise,  $m \text{ === } n$  is 1 if  $m$  equals  $n$  and 0 otherwise, and  $m - n$  is the intended value if  $m \geq n$  and  $\text{errPNat}$  that is an error otherwise.

Among the sorts used in the Minila specification are  $\text{PNat}$  for Peano natural numbers,  $\text{PNat\&Err}$  for Peano natural numbers and an error number,  $\text{Var}$  for Minila variables,  $\text{Exp}$  for Minila expressions,  $\text{Stm}$  for Minila statements,  $\text{Env}$  for environments (maps of  $\text{Var}$  to  $\text{PNat\&Err}$ , used to record assignments),  $\text{Env\&Err}$  for environments and an error environment,  $\text{Stack}$  for stacks,  $\text{Instr}$  for instructions,  $\text{Instr\&Err}$  for instructions and an error instruction, and  $\text{IList}$  for instruction sequences.  $\text{errEnv}$ ,  $\text{errStack}$ ,  $\text{errIList}$  and  $\text{errInstr}$  are the error environment, stack, instruction sequence and instruction, respectively. Stacks, instruction sequences and environments are specified as instances of  $\text{LIST}$  partially mentioned in the last section such that stacks are lists of  $\text{PNat\&Err}$ , instruction sequences are lists of  $\text{Instr\&Err}$  and environments are lists of  $\text{Var-PNat\&Err}$  pairs.  $\text{nil}$  is renamed  $\text{empStk}$  for stacks,  $\text{iln}$  for instruction sequences and  $\text{empEnv}$  for environments. Given an instruction list  $\text{il}$  and a natural number  $\text{pc}$ ,  $\text{nth}(\text{pc}, \text{il})$  returns the instruction pointed by  $\text{pc}$  in  $\text{il}$  if  $\text{pc}$  is within the bounds of  $\text{il}$  and  $\text{errInstr}$  otherwise. Given an environment  $\text{ev}$ , a variable  $\text{x}$  and a value  $\text{v}$  of  $\text{PNat\&Err}$ ,  $\text{lookup}(\text{ev}, \text{x})$  returns the value associated with  $\text{x}$  if any and  $\text{errPNat}$  otherwise, and  $\text{update}(\text{ev}, \text{x}, \text{v})$  returns the environment obtained by updating  $\text{ev}$  with  $\text{x}$  and  $\text{v}$ .

Let  $\text{N}$ ,  $\text{N1}$ ,  $\text{N2}$  and  $\text{PC}$  be CafeOBJ variables of  $\text{PNat}$ ,  $\text{NE}$  be one of  $\text{PNat\&Err}$ ,  $\text{V}$  be one of  $\text{Var}$ ,  $\text{E}$  be one of  $\text{Exp}$ ,  $\text{S}$ ,  $\text{S1}$  and  $\text{S2}$  be ones of  $\text{Stm}$ ,  $\text{EV}$  be one of  $\text{Env}$ ,  $\text{EE}$  be one of  $\text{Env\&Err}$ ,  $\text{I}$  be one of  $\text{Instr}$ ,  $\text{IL}$  be one of  $\text{IList}$  and  $\text{ST}$  be one of  $\text{Stack}$ .

The main part of the interpreter is the function `interpret` declared as follows:

```
op interpret : Stm -> Env&Err .
```

`interpret` takes a program (a statement) and returns a normal environment in which computation results are found if the program has been successfully interpreted and  $\text{errEnv}$  otherwise.

Excerpts from the set of equations defining `interpret` are shown:

```
eq interpret(S) = eval(S,empEnv) .
eq eval(S,errEnv) = errEnv .
eq eval((V := E ;),EV) = update(EV,V,evalExp(E,EV)) .
eq eval(if E {S1} else {S2},EV) = evalIf(evalExp(E,EV),S1,S2,EV) .
eq eval(while E {S},EV) = evalWhile(evalExp(E,EV),E,S,EV) .
eq eval(S1 S2,EV) = eval(S2,eval(S1,EV)) .
eq evalIf(errPNat, S1, S2, EV) = errEnv .
eq evalIf(N,S1,S2,EV) = if (0 < N) then {eval(S1,EV)} else {eval(S2,EV)} .
eq evalWhile(errPNat,E,S,EV) = errEnv .
eq evalWhile(N,E,S,EV)
  = if (0 < N) then {eval(while E {S},eval(S,EV))} else {EV} .
```

`evalExp` interprets an expression under an environment and returns a natural number or  $\text{errPNat}$ . The operator `if_then_{-}else_{-}` is prepared for each sort  $S$

such that the first argument is `Bool`, the second and third arguments are  $S$ , the second argument is selected if the first argument is `true` and the third argument is selected if the first argument is `false`.

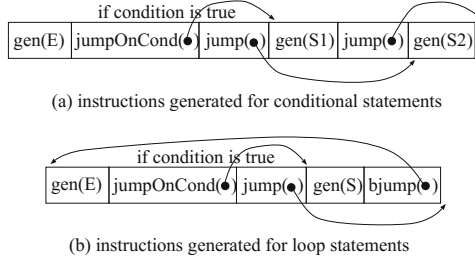
The virtual machine is a stack machine. Given an instruction sequence, it uses a program counter, a stack and an environment to execute the instruction sequence, and returns an environment or `errEnv`. The main part of the virtual machine is the function `run` declared as follows:

```
op run : IList -> Env&Err .
```

Excerpts from the set of equations defining `run` are shown:

```
eq run(IL) = exec(IL,0,empStk,empEnv) .
eq exec(IL,PC,ST,errEnv) = errEnv .
eq [FI]: exec(IL,PC,ST,EV) = exec2(nth(PC,IL),IL,PC,ST,EV) .
eq exec2(errInstr,IL,PC,ST,EV) = errEnv .
eq exec2(I,IL,PC,errPNat | ST,EV) = errEnv .
eq exec2(push(N),IL,PC,ST,EV) = exec(IL,s(PC),N | ST,EV) .
eq exec2(load(V),IL,PC,ST,EV) = exec(IL,s(PC),lookup(EV,V) | ST,EV) .
eq exec2(store(V),IL,PC,empStk,EV) = errEnv .
eq exec2(store(V),IL,PC,N1 | ST,EV) = exec(IL,s(PC),ST,update(EV,V,N1)) .
eq exec2(add,IL,PC,empStk,EV) = errEnv .
eq exec2(add,IL,PC,N1 | empStk,EV) = errEnv .
eq exec2(add,IL,PC,N2 | N1 | ST,EV) = exec(IL,s(PC),N1 + N2 | ST,EV) .
eq exec2(jump(N),IL,PC,ST,EV) = exec(IL,PC + N,ST,EV) .
eq exec2(bjump(0),IL,PC,ST,EV) = exec(IL,PC,ST,EV) .
eq exec2(bjump(s(N)),IL,0,ST,EV) = errEnv .
eq exec2(bjump(s(N)),IL,s(PC),ST,EV) = exec2(bjump(N),IL,PC,ST,EV) .
eq exec2(jumpOnCond(N),IL,PC,empStk,EV) = errEnv .
eq exec2(jumpOnCond(N),IL,PC,N1 | ST,EV)
  = if (N1 = 0) then {exec(IL,s(PC),ST,EV)} else {exec(IL,PC + N,ST,EV)} .
eq exec2(quit,IL,PC,ST,EV) = EV .
```

Among the instructions for the virtual machine are `push(N)`, `load(V)`, `store(V)`, `add`, `jump(N)`, `bjump(N)`, `jumpOnCond(N)` and `quit`. `push(N)` pushes  $N$  on the stack and increments the program counter. `load(V)` gets the value associated with  $V$  in  $EV$  by `lookup(EV,V)`, pushes the value on the stack and increments the program counter. If the stack is not empty, `store(V)` pops the stack, updates the environment with  $V$  and the value extracted from the stack and increments the program counter, and otherwise it leads to `errEnv`. If the stack contains at least two values, `add` pops the stack twice, pushes the addition of the two values extracted from the stack and increments the program counter, and otherwise it leads to `errEnv`. `jump(N)` increases the program counter by  $N$ . `bjump(N)` decreases the program counter by  $N$  if the program counter is greater than or equal to  $N$ , and otherwise it leads to `errEnv`. If the stack is not empty, `jumpOnCond(N)` pops the stack, increases the program counter by  $N$  if the value extracted from the stack is not zero and increments the program counter otherwise. If the stack is empty, `jumpOnCond(N)` leads to `errEnv`. `quit` returns the environment as the result of the execution.



**Fig. 1.** Instructions generated for conditional and loop statements

The main part of the compiler is the function `compile` declared as follows:

```
op compile : Stm -> ILIST .
```

Excerpts from the set of equations defining `compile` are shown:

```
eq compile(S) = gen(S) @ (quit | iln) .
eq gen(V := E ;) = genExp(E) @ (store(V) | iln) .
eq gen(if E {S1} else {S2}) = genExp(E) @ (jumpOnCond(s(s(0))) |
  (jump(s(s(len(gen(S1)))))) | (gen(S1) @ (jump(s(len(gen(S2)))) |
  gen(S2)))) .
eq gen(while E {S}) = genExp(E) @ ((jumpOnCond(s(s(0))) |
  jump(s(s(len(gen(S)))))) | gen(S)) @ (bjump(s(s(len(gen(S))
  + len(genExp(E)))))) | iln) .
eq gen(S1 S2) = gen(S1) @ gen(S2) .
```

Figure 1 shows the outlines of instructions generated for conditional and loop statements.

`interpret(Psr)` returns the following environment, and so does `run(compile(Psr))`:

```
(x , 2000000000000000) | (y , 141421356) |
(z , 141421356) | (tmp , 141421356) | empEnv
```

Therefore, we have successfully tested the Minila compiler correctness with respect to `Psr`. We can test the correctness with other terminating Minila programs. To conclude the Minila compiler correctness with respect to all terminating Minila programs, however, we need to formally prove that for all terminating Minila programs  $p$   $\text{interpret}(p) = \text{run}(\text{compile}(p))$ .

## 4 Formal Verification

What we would like to do is to formally verify the correctness of the Minila compiler for all Minila programs. To this end, we need to formalize the correctness. We basically define it by using the Minila interpreter as an oracle:  $(\forall S : \text{Stm}) \text{run}(\text{compile}(S)) = \text{interpret}(S)$ . However, programs may not terminate. If



that is the case, the formula does not hold. Then, we only take into account terminating programs. Therefore, we define the correctness (CompCorr) of the Minila compiler as follows:

$$(\forall S:\text{Stm}) \text{tc}(S) \Rightarrow \text{run}(\text{compile}(S)) = \text{interpret}(S)$$

where  $\text{tc}(S)$  says that  $S$  terminates.  $\text{tc}$  is defined as follows:  $\text{tc}(S) = \text{tc}(S, \text{empEnv})$ .  $\text{tc}(S, \text{EE})$  is defined as follows:

```

eq tc(S, errEnv) = true .
eq tc(estm, EV) = true .
eq tc(V := E ;, EV) = true .
ceq tc(if E {S1} else {S2}, EV) = true if evalExp(E, EV) = errPNat .
ceq tc(if E {S1} else {S2}, EV) = tc(S1, EV) if 0 < evalExp(E, EV) .
ceq tc(if E {S1} else {S2}, EV) = tc(S2, EV) if evalExp(E, EV) = 0 .
ceq tc(S1 S2, EV) = tc(S2, eval(S1, EV)) if tc(S1, EV) .

```

To complete the definition of  $\text{tc}$ , we need to tackle the case “while  $E \{S\}$ ,” which requires to define the new function as follows:

```

op eval : PNat&Err Stm Env&Err -> Env&Err {strat (0 1 0)}
eq eval(errPNat, S, EV) = errEnv .
eq eval(N, S, errEnv) = errEnv .
eq eval(0, S, EV) = EV .
eq eval(S, eval(N, S, EE)) = eval(s(N), S, EE) .

```

The LHS and RHS of the last equation are swapped because of a verification purpose. We also need to give the strategy (0 1 0) to  $\text{eval}$  to control rewriting [6, 7].

$\text{tc}(\text{while } E \{S\}, \text{EV})$  is defined as follows:

$$(\exists N:\text{PNat}) \text{tc}(N, E, S, \text{EV})$$

where  $\text{tc}(N, E, S, \text{EV})$  is defined as follows:

```

(evalExp(E, eval(N, S, EV)) = 0  $\vee$  evalExp(E, eval(N, S, EV)) = errPNat)  $\wedge$ 
( $\forall K:\text{PNat}$ ) (K < N  $\Rightarrow$  tc(S, eval(K, S, EV))  $\wedge$  K < N  $\Rightarrow$  0 < evalExp(E, eval(K, S, EV)))

```

The most important lemma (Lem1) is as follows:

$$(\forall S:\text{Stm})(\forall \text{IL1}, \text{IL2}:\text{IList})(\forall \text{ST}:\text{Stack})(\forall \text{EE}:\text{Env\&Err})$$

$$\text{tc}(S, \text{EE}) \Rightarrow \text{eval}(\text{IL1} @ \text{gen}(S) @ \text{IL2}, \text{len}(\text{IL1}), \text{ST}, \text{EE}) =$$

$$\text{eval}(\text{IL1} @ \text{gen}(S) @ \text{IL2}, \text{len}(\text{IL1} @ \text{gen}(S)), \text{ST}, \text{eval}(S, \text{EE}))$$

The LHS of the conclusion part of Lem1 is the case in which  $\text{IL1}$  has been processed. Therefore, the program counter is  $\text{len}(\text{IL1})$  and  $\text{EE}$  is the environment obtained by processing  $\text{IL1}$  under  $\text{empEnv}$ . The RHS is the case in which  $\text{gen}(S)$  has been processed in the former case. Hence, the program counter is  $\text{len}(\text{IL1} @ \text{gen}(S))$  and the environment is  $\text{eval}(S, \text{EE})$ . Lem1 says that if  $S$  terminates in  $\text{EE}$ , both cases are equal.

In addition to Lem1, we need some more lemmas to complete the formal verification. One of those lemmas is

$$(\forall S:\text{Stm})(\forall \text{IL1}, \text{IL2}:\text{IList}) \text{nth}(\text{len}(\text{LI1} @ \text{gen}(S)), \text{LI1} @ \text{gen}(S) @ (\text{I} | \text{LI2})) = \text{I}$$

Let the lemma be referred as Lem2. Let  $n$  be  $\text{len}(\text{LI1} @ \text{gen}(S))$ . Lem2 says that the  $n$ th instruction in  $\text{LI1}@\text{gen}(S)@(I|\text{LI2})$  is  $\text{I}$ .

The following corollaries can be derived from Lem1 and Lem2:

$$\begin{aligned} & \text{tc}(S, \text{EE}) \Rightarrow \\ & \text{eval}(\text{gen}(S) @ \text{IL2}, 0, \text{ST}, \text{EE}) = \text{eval}(\text{gen}(S) @ \text{IL2}, \text{len}(\text{gen}(S)), \text{ST}, \text{eval}(S, \text{EE})) \\ & \text{nth}(\text{len}(\text{gen}(S)), \text{gen}(S) @ (\text{I} | \text{LI2})) = \text{I} \end{aligned}$$

Note that quantifiers are omitted. The corollaries are referred as CorLem1 and CorLem2.

CompCorr is proved by case splitting based on  $\text{tc}(s, \text{empEnv})$ . The case is split into (1) in which it is **true** and (2) in which it is **false**. Case (1) is further split into (1.1) in which  $\text{eval}(s, \text{empEnv})$  is a normal environment and (1.2) in which it is **errEnv**. The proof score of case (1.1) is as follows:

```
open VERIFY-COMP .
-- FRESH CONSTANTS
op s : -> Stm .
op ev : -> Env .
-- CASE SPLITTING HYPOTHESES
eq tc(s, empEnv) = true .
eq eval(s, empEnv) = ev .
-- LEMMAS
ceq eval(gen(S) @ IL2, 0, ST, EE)
  = eval(gen(S) @ IL2, len(gen(S)), ST, eval(S, EE)) if tc(S, EE) .
eq nth(len(gen(S)), gen(S) @ (I | LI2)) = I .
-- CHECK
red tc(s) implies run(compile(s)) = interpret(s) .
close
```

CorLem1 is written as a conditional equation. The proof score of case (1.2) is obtained by replacing the RHS of the second case splitting hypothesis equation with **errEnv**. The proof score of case (2) is obtained by replacing the RHS of the first case splitting hypothesis equation with **false** and deleting the other three equations.

Lem1 is proved by structural induction on  $S$ . In the induction case in which “while  $e \{e\}$ ” is taken into account, what to do is to show<sup>4</sup>

$$\begin{aligned} & (\forall N:\text{PNat})(\forall EV:\text{Env}) (\forall \text{IL1}, \text{IL2}:\text{IList})(\forall \text{ST}:\text{Stack}) \\ & (\text{tc}(N, e, s, EV) \Rightarrow \text{concl}(e, s, EV, \text{IL1}, \text{IL2}, \text{ST})) \end{aligned}$$

<sup>4</sup>  $(\forall y : T2)(\forall z : T3)((\exists x : T1)p(x, y) \Rightarrow q(y, z))$  is equivalent to  $(\forall x : T1)(\forall y : T2)(\forall z : T3)(p(x, y) \Rightarrow q(y, z))$ .

where  $\text{concl}(E, S, EV, IL1, IL2, ST)$  is as follows:

```
eval(IL1 @ gen(while E {S}) @ IL2, len(IL1), ST, EV) =
eval(IL1 @ gen(while E {S}) @ IL2, len(IL1 @ gen(while E {S})),
    ST, eval(while E {S}, EV))
```

The induction hypothesis is

```
(∀ EV:Env)(∀ IL1, IL2:IList)(∀ ST:Stack) (tc(s, EV) ⇒ prem(s, EV, IL1, IL2, ST))
```

where  $\text{prem}(S, EV, IL1, IL2, ST)$  is as follows:

```
eval(IL1 @ gen(S) @ IL1, len(IL1), ST, EV) =
eval(IL1 @ gen(S) @ IL2, len(IL1 @ gen(S)), ST, eval(S, EV))
```

To tackle the induction case, we need several lemmas, one of which as follows:

```
(∀ N:PNat)(∀ E:Exp)(∀ S:Stm)(∀ EV:Env) (∀ IL1, IL2, IL3, IL4:IList)(∀ ST:Stack)
prems(S, EV, IL3, IL4, ST) ⇒ tc(N, E, S, EV) ⇒ concl(E, S, EV, IL1, IL2, ST)
```

The lemma is referred as Lem3. Lem3 is proved as follows: assuming the premise  $\text{prems}(s, ev, IL3, IL4, st)$ ,  $\text{tc}(N, E, s, ev) \Rightarrow \text{concl}(E, s, ev, IL1, IL2, st)$  is proved by structural induction on  $N$ . The induction case is split into two cases (1) in which  $\text{evalExp}(e, \text{eval}(s(n), s, ev))$  is 0 and (2) in which it is  $\text{errPNat}$ . Then, case (1) is further split into two cases (1.1) in which  $\text{eval}(s, ev)$  is a normal environment and (2.2) in which it is  $\text{envErr}$ , and so is case (2).

The proof score of case (1.1) is as follows<sup>5</sup>:

```
open VERIFY-COMP .
-- Import some modules.
pr(EVAL) pr(DEL)
```

The module `VERIFY-COMP` is opened and the two module `EVAL` and `DEL` are imported. `VERIFY-COMP` contains the Minila interpreter, compiler and virtual machine and some necessary things to express `CompCorr` and lemmas. `EVAL` and `DEL` contain some auxiliary things for lemmas.

```
-- Fresh constants
ops n k a b : -> PNat . op e : -> Exp . op s : -> Stm . op ev : -> Env .
ops il1 il2 : -> IList . op stk : -> Stack .
```

Fresh constants are declared, representing arbitrary values of the corresponding sorts.

```
-- Variants of prem(s, ev, IL3, IL4, stk)
eq exec2(nth(0, gen(s) @ IL4), IL3 @ gen(s) @ IL4, len(IL3), stk, ev)
= exec2(nth(0, IL4), IL3 @ gen(s) @ IL4, len(IL3 @ gen(s)), stk, eval(s, ev)) .
eq exec2(hd(gen(s) @ IL4), IL3 @ gen(s) @ IL4, len(IL3), stk, ev)
= exec2(nth(0, IL4), IL3 @ gen(s) @ IL4, len(IL3 @ gen(s)), stk, eval(s, ev)) .
```

<sup>5</sup> Some descriptions on the proof score are inserted.

Two variants of the premise  $\text{prem}(s, \text{ev}, \text{IL3}, \text{IL4}, \text{stk})$  are declared.

```
-- tc(s(n), e, s, ev)
ceq evalExp(e, eval(K, s, ev)) = 0 if K = s(n) .
ceq evalExp(e, eval(K, s, ev)) = s(a) if K < s(n) .
ceq tc(s, eval(K, s, ev)) = true if K < s(n) .
```

The premise  $\text{tc}(s(n), e, s, \text{ev})$  of what to prove is assumed, which is written as equations.

```
-- Induction Hypothesis (IH)
-- Check the premise of IH
eq (k < n) = true .
red evalExp(e, eval(n, s, eval(s, ev))) = 0 .
red 0 < evalExp(e, eval(k, s, eval(s, ev))) .
red tc(s, eval(k, s, eval(s, ev))) .
```

We check if the premise of the induction hypothesis holds. Since each of the three reductions is true, the premise holds.

```
-- The conclusion of IH
eq [IH]: exec(il1 @ gen(while e {s}) @ il2, len(il1), stk, eval(s, ev))
  = exec(il1 @ gen(while e {s}) @ il2, len(il1 @ gen(while e {s})), stk,
    eval(while e {s}, eval(s, ev))) .
```

Therefore, the conclusion of the induction hypothesis holds, which is written as an equation.

```
-- Lemmas
eq exec(genExp(E) @ IL2, 0, ST, EE)
  = exec(genExp(E) @ IL2, len(genExp(E)), evalExp(E, EE) | ST, EE) .
eq exec2(hd(genExp(E) @ IL2), IL1 @ genExp(E) @ IL2, len(IL1), ST, EV)
  = exec2(nth(0, IL2), IL1 @ genExp(E) @ IL2, len(IL1 @ genExp(E)),
    evalExp(E, EV) | ST, EV) .
eq exec2(bjump(len(IL1) + N), IL, len(IL1) + PC, ST, EV)
  = exec2(bjump(N), IL, PC, ST, EV) .
eq exec2(bjump(len(IL1)), IL, PC + len(IL1), ST, EV)
  = exec2(bjump(0), IL, PC, ST, EV) .
eq exec2(bjump(len(IL1)), IL, len(IL1), ST, EV) = exec2(bjump(0), IL, 0, ST, EV) .
eq len(IL2 @ IL1) = len(IL1) + len(IL2) .
eq nth(N, IL) = hd(del(N, IL)) .
```

We use several lemmas. The first two equations are derived from the following lemma:

$$(\forall E:\text{Exp})(\forall \text{IL1}, \text{IL2}:\text{IList})(\forall \text{ST}:\text{Stack})(\forall \text{EE}:\text{Env\&Err})$$

$$\text{exec}(\text{IL1} @ \text{genExp}(E) @ \text{IL2}, \text{len}(\text{IL1}), \text{ST}, \text{EE}) = \text{exec}(\text{IL1} @ \text{genExp}(E) @ \text{IL2},$$

$$\text{len}(\text{IL1} @ \text{genExp}(E)), \text{evalExp}(E, \text{EV}) | \text{ST}, \text{EE})$$

The lemma is proved by structural induction on  $E$ . The next three equations are lemmas about the instruction `bjump`. The next two equations are basic lemmas about standard functions of lists.

```

-- Check the value of evalExp(e, ev)
red evalExp(e, ev) = evalExp(e, eval(0, s, ev)) .
red 0 < s(m) .
-- Therefore, 0 < evalExp(e, ev)
eq evalExp(e, ev) = s(b) .

```

Since each of the two reductions is true,  $\text{evalExp}(e, \text{ev})$  is greater than 0, represented by  $s(b)$ .

```

-- Case splitting hypothesis
op ev2 : -> Env .
eq eval(s, ev) = ev2 .

```

We assume  $\text{eval}(s, \text{ev})$  is a normal environment.

```

-- Check (part 1)
-- 3-stepped verification to show
-- exec(il1 @ gen(while e {s}) @ il2, len(il1), stk, ev)
-- = exec(il1 @ gen(while e {s}) @ il2, len(il1), stk, eval(s, ev))

```

Since the set of equations used as a set of left-to-right rewrite rules is not confluent, what we would like to show here does not reduce to `true`. So, it is shown in three steps. An outline of the three stepped verification is as follows. Let us suppose that we would like to show that  $lhs$  equals  $rhs$  but  $lhs = rhs$  does not reduce to true. Instead of reducing  $lhs = rhs$ , we reduce (1)  $lhs = f(a)$ , (2)  $a = b$  and (3)  $f(b) = rhs$ . If all three terms reduce to true, we successfully show  $lhs = rhs$ . It does not suffice, however, to divide the reduction into the three steps because some equations cannot be applied as desired. We need to control reductions so that the lemma can be applied. To this end, we give the strategy (0 1 2 3 4 5 0) to `exec2`.

```

op pca : -> PNat . ops ila il1b il2b : -> IList .
eq pca = s(s((len(il1) + len(genExp(e)))))) .
eq ila
  = (il1 @ (genExp(e) @ (jumpOnCond(s(s(0))) | (jump(s(s(len(gen(s))))
    | (gen(s) @ (bjump(s(s((len(gen(s)) + len(genExp(e)))) | il2))))))) .
eq il1b = il1 @ (genExp(e) @ (jumpOnCond(s(s(0)))
  | (jump(s(s(len(gen(s)))) | iln))) .
eq il2b = (bjump(s(s((len(gen(s)) + len(genExp(e)))) | il2)) .

```

Some constants are prepared as abbreviations.

```

-- step 1 : lhs = f(x)
red exec(il1 @ gen(while e {s}) @ il2, len(il1), stk, ev)
  = exec2(hd(gen(s) @ il2b), il1b @ gen(s) @ il2b, pca, stk, ev) .
-- step 2 : x = y
red ila = il1b @ gen(s) @ il2b .
red len(il1b) = pca .
-- step 3 : f(y) = rhs
red exec2(hd(gen(s) @ il2b), il1b @ gen(s) @ il2b, len(il1b), stk, ev)
  = exec(il1 @ gen(while e {s}) @ il2, len(il1), stk, eval(s, ev)) .

```

Each of the reductions is true.

```
-- From the 3 stepped verification
eq [Check1]: exec(il1 @ gen(while e {s}) @ il2,len(il1), stk, ev)
  = exec(il1 @ gen(while e {s}) @ il2,len(il1),stk,eval(s,ev)) .
```

Therefore, we have the equation Check1.

```
-- Check (part 2)
red exec(il1 @ gen(while e {s}) @ il2,len(il1 @ gen(while e {s})),stk,
  eval(while e {s},eval(s,ev)))
  = exec(il1 @ gen(while e {s}) @ il2,len(il1 @ gen(while e {s})),stk,
  eval(while e {s},ev)) .
```

The reduction is true.

```
-- From the reduction
eq [Check2]: exec(il1 @ gen(while e {s}) @ il2,
  len(il1 @ gen(while e {s})),stk,eval(while e {s},eval(s,ev)))
  = exec(il1 @ gen(while e {s}) @ il2,
  len(il1 @ gen(while e {s})),stk,eval(while e {s},ev)) .
```

Hence, we have the equation Check2.

```
-- Check (final)
start exec(il1 @ gen(while e {s}) @ il2,len(il1),stk,ev) =
  exec(il1 @ gen(while e {s}) @ il2,len(il1 @ gen(while e {s})),stk,
  eval(while e {s},ev)) .
apply Check1 at (1) .
apply IH at (1) .
apply Check2 at (1) .
apply reduce at term .
close
```

Finally, case (2.1) is discharged. CafeOBJ allows human users to apply a specific equation to a specific position with the `apply` command in a given term by the `start` command. Case (1.2) can be discharged likewise, and so can the two cases of case (2). The base case can also be discharged.

Lem3 can be used to discharge the induction cases in which “`while e {s}`” is taken into account when Lem1 is proved by structural induction on  $S$ . The other induction cases and the base case can also be discharged likewise. Moreover, all necessary lemmas can be proved as well. Therefore, we have proved CompCorr.

## 5 Discussion

Since Minila is simple, our formal verification of the Minila compiler (and virtual machine) is a tiny step towards formal verification of real life compilers that has been tackled in CompCert [8,9] ([compcert.inria.fr](http://compcert.inria.fr)). As described, compilers heavily affect the correctness of software and then formal verification of compilers

is one of the key technologies that make it possible to make our future IoT era highly reliable. We do not think that it suffices to have one technique for formal verification of compilers. We need to develop multiple such techniques so as to make formal compiler verification techniques mature enough. Therefore, we believe that our formal verification of the Minila compiler (and virtual machine) is an important step towards the goal as well.

Programming language semantics in algebraic specification languages has originated in the Goguen and Malcolm's Algebraic Semantics of Imperative Programs (ASIP) [10], from which K is descended. OBJ3 [11], the predecessor of CafeOBJ, has been used for ASIP. K [12–14] ([www.kframework.org](http://www.kframework.org)) is a rewrite-based executable semantic framework dedicated to description of programming language semantics. Several real-world programming languages' semantics, such as ANSI C, Java and Python, has been described in K. K has been implemented on the top of Maude [15] ([maude.cs.illinois.edu](http://maude.cs.illinois.edu)) that is a sibling language of CafeOBJ. Equations are main ingredients in ASIP, while rewrite rules are in K. Our approach as well as K is also descended from ASIP and uses equations as main ingredients. K provides several analysis techniques for programming language semantics, but to our knowledge K has not been used to formally verify any compilers. To extend our approach to make it possible to formally verify real life compilers, such as C compilers and Java compilers, one possible approach is to adopt techniques used in K to describe semantics of real life programming languages. Since both K and our approach are descended from ASIP, sharing algebraic techniques matured in the course of developments of algebraic specification languages, such as CafeOBJ and Maude, this approach seems promising as well as feasible.

What if the Minila interpreter contains flaws? Our formal verification implies that the Minila compiler and virtual machine contain the same flaws. The Minila interpreter is the semantics of Minila in the sense of ASIP, the requirements that must be satisfied by the Minila compiler and virtual machine, and must be carefully developed. The issue is shared by validation of formal specifications that is to confirm that formal specifications really faithfully capture requirements or clients' intentions. Although the issue is very important in software development, it is not that simple and the very crucial research topic that must be tackled by engineers as well as researchers.

We assume that programs terminate to formally verify the Minila compiler. Note that we do not need to verify programs terminate. The main proof technique we have used is structural induction. If we do not assume the termination, we need to deal with infinity, which is beyond the scope in which structural induction can be used. We need to utilize coalgebra and coinduction [16] to deal with infinity. Fortunately, CafeOBJ has adopted hidden algebra [17, 18] as one of CafeOBJ logics, which is close to coalgebra and makes it possible to deal with infinity.

The proof technique used in CompCert is to show that any safe programs in a source language can be simulated by the compiled programs in a target language. The same technique has been used to formally verify a compiler for a subset of

Java with Isabelle/HOL ([www.cl.cam.ac.uk/research/hvg/Isabelle/](http://www.cl.cam.ac.uk/research/hvg/Isabelle/)) [19]. Their approach uses state machines as the semantics of both source and target programming languages, making it possible to deal with non-terminating programs. We will carefully investigate their approach, making a comparison between theirs and ours and clarifying the pros and cons of each approach.

The second author's approach [1] assumes that if a given program  $S$  is non-terminating, `interpret(S)` returns a constant `env-non-halt`, which is used to define the correctness of the Minila compiler. His specification of `interpret` in CafeOBJ, however, does not return `env-non-halt` even if a clearly non-terminating program, such as `while 0 == 0 { x := 0 ; }`, is given. Therefore, although both the first and second authors' approaches are quite similar, the assumption used in the second author's approach would not be very reasonable.

Kokichi Futatsugi gave a tutorial on CafeOBJ [20] organized by the Compilers and Languages Group at the Institute of Computer Languages, the Vienna University of Technology in 2012. He talked about CafeOBJ specifications of Minila interpreter, compiler and virtual machine, and the formal correctness verification of part of the Minila compiler that was an arithmetic expression compiler<sup>6</sup>, but not the Minila compiler described in the present paper.

## 6 Conclusion

The paper has reported on theorem proving the Minila compiler correctness for all terminating Minila programs, meaning that for all terminating Minila program  $p$  the result of interpreting  $p$  with the interpreter is the same as the one of processing by the virtual machine the instruction sequence generated from  $p$  by the compiler. The proof has been conducted by writing proof scores in CafeOBJ and executing them with the CafeOBJ system. Specifications in CafeOBJ are executable by using equations as left-to-right rewrite rules and proof scores are those in CafeOBJ. The interpreter, the virtual machine and the compiler written in CafeOBJ can be used as their prototype implementations and therefore can be tested, such that for any concrete terminating Minila program  $p$  it is doable to test that the result of interpreting  $p$  with the interpreter is the same as the one of processing by the virtual machine the instruction sequence generated from  $p$  by the compiler. It is one strong point of CafeOBJ's that specifications written in CafeOBJ can be used as ordinary executable programs, making it possible to test them. Some possible future directions have been mentioned in the last section.

## References

1. Bao, N.Q.T.: Verifying the correctness of compiler for an imperative programming language. Master's thesis, School of Information Science, JAIST (2011)
2. Daudier, D.: Verification of compilers based on algebraic specifications. Technical report, ISAE ENSMA (2017)

<sup>6</sup> The slides used are available at <http://www.jaist.ac.jp/~kokichi/class/TUW1207+08/>.



3. Morgan, O.: C compiler validation for embedded targets - qualifying compilers for use in safety-critical projects. A white paper of Solid Sands B.V. (2016)
4. Diaconescu, R., Futatsugi, K.: CafeOBJ Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. AMAST Series in Computing, vol. 6. World Scientific, Singapore (1996)
5. Goguen, J.A.: Proving and rewriting. In: Kirchner, H., Wechler, W. (eds.) ALP 1990. LNCS, vol. 463, pp. 1–24. Springer, Heidelberg (1990). [https://doi.org/10.1007/3-540-53162-9\\_27](https://doi.org/10.1007/3-540-53162-9_27)
6. Futatsugi, K., Goguen, J.A., Jouannaud, J.P., Meseguer, J.: Principles of OBJ2. In: 12th ACM SIGPLAN-SIGACT POPL, pp. 52–66. ACM (1985)
7. Ogata, K., Futatsugi, K.: Operational semantics of rewriting with the on-demand evaluation strategy. In: 15th ACM SAC, pp. 756–764. ACM (2000)
8. Leroy, X.: A formally verified compiler back-end. *J. Autom. Reason.* **43**, 363–446 (2009)
9. Leroy, X.: Formal verification of a realistic compiler. *Commun. ACM* **52**, 107–115 (2009)
10. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs. The MIT Press, Cambridge (1996)
11. Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K.: Introducing OBJ. In: Goguen, J., Malcolm, G. (eds.) Software Engineering with OBJ. ADFM, vol. 2, pp. 3–167. Springer, Heidelberg (2000). [https://doi.org/10.1007/978-1-4757-6541-0\\_1](https://doi.org/10.1007/978-1-4757-6541-0_1)
12. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. *J. Logic Algebraic Program.* **79**, 397–434 (2010)
13. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: 36th ACM SIGPLAN PLDI, pp. 336–345. ACM (2015)
14. Bogdănaş, D., Roşu, G.: K-Java: a complete semantics of Java. In: 42nd ACM SIGPLAN-SIGACT POPL, pp. 445–456. ACM (2015)
15. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71999-1>
16. Jacobs, B., Rutten, J.: An introduction to (co)algebras and (co)induction. In: Advanced Topics in Bisimulation and Coinduction, pp. 38–99 (2011)
17. Goguen, J.A., Malcolm, G.: A hidden agenda. *Theoret. Comput. Sci.* **245**, 55–101 (2000)
18. Diaconescu, R., Futatsugi, K.: Behavioural coherence in object-oriented algebraic specification. *J. Univers. Comput. Sci.* **6**, 74–96 (2011)
19. Strecker, M.: Formal verification of a Java compiler in Isabelle. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 63–77. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45620-1\\_5](https://doi.org/10.1007/3-540-45620-1_5)
20. Futatsugi, K.: Introduction to specification and verification in CafeOBJ (2012). <http://www.informatik.tuwien.ac.at/news/630>

## Author Index

- Cheng, Zhuo 51  
Cui, Jin 148
- Daudier, Dorian 200  
Duan, Zhenhua 148
- Jiang, Zhouxian 123
- Li, Guangyuan 111  
Li, Guoqiang 69  
Li, Honghui 123  
Li, Siyuan 187  
Liang, Hongliang 111  
Lin, Chin-Fu 39  
Lin, Chung-Ling 111  
Liu, Shaoying 24, 171, 187  
Lu, Yonggang 51
- Nagoya, Fumiko 24  
Navrátil, Ondřej 39  
Ngoc Quoc Bao, Trinh 200  
Nguyen, Tam Thi Thanh 3
- Ogata, Kazuhiro 3, 200
- Peng, Sheng-Lung 39
- Ren, Liyuan 133
- Shen, Wuwei 111  
Shu, Xinfeng 88, 133
- Tao, Xiuting 69  
Tian, Cong 148  
Tian, Xuetao 123
- Wang, Meng 148  
Wang, Mengnan 88  
Wang, Xiaobing 88, 133
- Xue, Jianxin 51
- Zhang, Haitao 51  
Zhang, Nan 148  
Zhao, Liang 133  
Zhao, Pan 171