GRACE TECHNICAL REPORTS

Proceedings of the Sixth Asian Workshop on Foundations of Software

Zhenjiang HU and Jian ZHANG
(Editors)

GRACE-TR 2009–01 April 2009

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

The GRACE technical reports are published as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. Copyright and all rights therein are maintained by the authors or by other copyright holders, notwithstanding that they have offered their works here electronically. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each author's copyright. These works may not be reposted without the explicit permission of the copyright holder.

Preface

This technical report contains the proceedings of the *Sixth Aisan Workshop on Foundations of Software* (AWFS 2009) held in Tokyo, Japan, April 6-8, 2009, hosted by the GRACE Center of National Institute of Informatics.

The Asian Workshop on Foundations of Software (AWFS) addresses foundational problems in current and future software design, development, and analysis. The previous five AWFS were held in Hangzhou in 2002, Nanjing in 2003, Xi'an in 2004, Beijing in 2005, and Xiamen in 2007. It is our best wish that this workshop would further stimulate various activities leading to formation of new forums for Asian researchers in the area of software science and technology.

This year, we have three invited talks given by David Lorge Parnas, Yike Guo, and Shinichi Honiden, nine presentations of regular papers (including three short papers) selected from the submissions, eight short presentations of published papers, and six short presentations of ongoing work.

On behalf of the program committee, we would like to thank Dongming Wang and Tetsuo Ida, the program chairs of AWFS 2007, for their invaluable advices on organization of AWFS 2009, the invited speakers who agreed to give talks, all those who submitted papers, and all the referees for their careful work in the reviewing and selection process. The support of our sponsors is also gratefully acknowledged. In addition to the GRACE Center of National Institute of Informatics, we are indebted to Asian Association for Foundation of Software (AAFS). Finally, we would like to thank the members of the local arrangements committee, notably Yoshiko Asano, Akimasa Morihata, Yingfei Xiong, and Yumi Yamasaki, for their invaluable support throughout the preparation and organization of the symposium.

April 2009

Zhenjiang Hu
Jian Zhang

Workshop Organization

Workshop Chair

Tetsuo Ida	Tsukuba University, Japan

Program Chairs

Zhenjiang Hu	National Institute of Informatics, Japan
Jian Zhang	Chinese Academy of Sciences, China

Program Committee

Yiyun Chen	University of Science and Technology of China, China
Wei-Ngan Chin	National University of Singapore, Singapore
Jin Song Dong	National University of Singapore, Singapore
Yuxi Fu	Shanghai Jiaotong University, China
Qingshan Jiang	Xiamen University, China
Xuandong Li	Nanjing University, China
Shaoying Liu	Hosei University, Japan
Zhiming Liu	UNU/IIST, China
Shilong Ma	Beihang University, China
Shin-Cheng Mu	Academia Sinica, Taiwan
Yasuhiko Minamide	Tsukuba University, Japan
Mizuhito Ogawa	JAIST, Japan
Atsushi Ohori	Tohoku University, Japan
Nguyen Hua Phung	Ho Chi Minh City Univ. of Technology, Vietnam
Zongyan Qiu	Peking University, China
Masahiko Sato	Kyoto University, Japan
Zhong Shao	Yale University, USA
Masato Takeichi	University of Tokyo, Japan
Dongming Wang	Beihang University, China and UPMC-CNRS, France
Ji Wang	National University of Defense Technology, China
Yi Wang	Uppsala University, Sweden
Kwangkeun Yi	Seoul National University, Korea
Mingsheng Ying	Tsinghua University, China
Nobukazu Yoshioka	National Institute of Informatics, Japan
Taiichi Yuasa	Kyoto University, Japan
Wenhui Zhang	Chinese Academy of Sciences, China
Jianjun Zhao	Shanghai Jiaotong University, China

Local Arrangements

Yoshiko Asano	National Institute of Informatics, Japan
Akimasa Morihata	University of Tokyo, Japan
Yingfei Xiong	University of Tokyo, Japan
Yumi Yamasaki	National Institute of Informatics, Japan

Table of Contents

Session 4: Languages

Session 5: AOP and Program Transformation

Session 6: Logic and Formal Method

Invited Talk III

Session 7: Language Implementation

Session 8: Programming Algebra

Functional Documentation Using Tabular Expressions - An Integrated Approach to the Use of Mathematics in Computer System Design

David Lorge Parnas

Emeritus Professor
McMaster University
Canada

Abstract
Over a period of more than 40 years, Computer Scientists have been proposing mathematical approaches to software development. We have seen countless papers about:

- Description and analysis of networks of hardware elements (logic design)
- Specification and verification of sequential programs.
- Specification and verification of concurrently executing programs
- Specification and description of module and component interfaces
- Description of a module or components internal design decisions
- Description/Specification of a component or modules external behaviour
- Description/Specification of a computer systems external behaviour

In this talk I present an approach that can be applied to all of these problems. It allows designers to use the same notation throughout and to check on the correctness of the results at the end of each design phase. The functional approach defines the content of each document. Tabular expressions have the precision of conventional mathematics but present information in a way that is easy for developers to use.

A Global-to-Local Approach to Rigorously Developing Distributed System with Exception Handling

(Full version is published elsewhere. Included here for informal use only)

Chao Cai, Zongyan Qiu
LMAM and Department of Informatics, School of Math.,
Peking University, Beijing, China
E-mail: {caic, qzy}math.pku.edu.cn

Abstract

Cooperative distributed system covers a wide range of applications such as systems for industrial controlling and business-to-business trading, which are usually safety-critical. Coordinated exception handling (CEH) refers to exception handling in cooperative distributed system, where exceptions raised on a peer should be coped with by all relevant peers in a consistent manner. A crucial problem in using a CEH algorithm is how to develop peers which are guaranteed coherent in both normal and exceptional executions. Straightforward testing or model checking is very expensive. In this paper, we propose an effective way to develop systems with correct CEH behavior. Firstly, we formalize a CEH algorithm by proposing a Peer Process Language to precisely describe distributed systems and their operational semantics. Then we dig out a set of syntactic conditions, and prove its sufficiency to ensure system coherence. Finally, we propose a global-to-local approach, including a language for describing distributed systems from a global perspective and a projection algorithm, for developing the systems. Given a well-formed global description, a set of peers can be generated automatically. We prove the system composed by these peers satisfies the conditions, that is, it is always coherence and correct with respect to CEH.

Keywords: Distributed system, Exception Handling, Fault Tolerant, Formal Methods

1 Introduction

A cooperative distributed system involves several independent peers (subsystems) that work together to implement some global function or achieve some common goal. Many application systems fall into this category, e.g., industrial controlling system [24], patients' embedded accessorial system [5], railway scheduler [2], and business-to-business trading system. Many of these systems are safety-critical, which may cause risks for human lives or great financial losses if not fault-tolerant. Thus, it is crucial to guarantee that they provide intended behavior even when some faults or errors occur in execution.

To achieve fault-tolerance, exception handling (EH) mechanism is often used as a recovery-layer. With EH support, any perceptible fault or error are converted to some exception. The language or platform offers a kind of control structures that allows programmers to describe the replacement of the normal execution when exceptions occur. Most modern programming languages include EH mechanism as a fundamental feature, for easily describing the application-specific logic of the recovery procedures.

Although the EH mechanism for sequential programs is relative mature, it is not the case for the distributed systems. Due to the decentralization and cooperation features, in considering the exception handling here, we will inevitably meet two fundamental new problems: *coordinated exception handling* and *concurrent exceptions*.

Coordinated exception handling (CEH) refers to the situation where an exception occurs on one peer but cannot be dealt with by the peer locally. In this case, the exception should be

propagated to all relevant peers so that they can work collaboratively for returning the whole system to a consistent state [4, 1]. Various reasons call for attention on *concurrent exceptions* in CEH. Because peers in a system run parallelly and independently, two or more exceptions may occur on different peers "simultaneously". A typical situation is when a peer meets an exception and notifies its partners for this, some partners may encounter other exception(s) before receiving the message. For the coordinated handling to be possible, those situations should be integrated into the cooperation model.

Randell and Xu *et al* [4, 22] developed an algorithm for exception handling in cooperative distributed systems, based on the concept of "Coordinated Atomic" (CA) actions. The algorithm copes with both problems mentioned above. However, the algorithm was informal described . To introduce a linguistic mechanism is an important future work announced in [22]. One motivation of the work presented here is to make the algorithm clearly defined and easy-usable.

A crucial problem in using the CEH algorithms is how to develop peers which are guaranteed coherent in both normal and exceptional execution. On the one hand, a composed system consists of a set of cooperative peers, where each peer participates several CAs and may encounter exceptions anywhere in the execution. On the other hand, exceptions may be propagated in two directions: either horizontally to other peers working in the same CA, or vertically to the enclosing CA. Both situations make it very difficult to guarantee the system coherency. Straightforward testing or model checking is very expensive. As shown in [21], checking a manufacture controlling system generates 10^{34} states, and costs more than a week. As another motivation of our work, we want to develop a new approach not only to rigorously develop distributed system with exception handling, but also to provide effective static verification.

The first contribution we made is the definition of a Peer Process Language (PPL), by which we can formally define *peers* and the *composed systems*. We give an operational semantics for PPL, and embed the CEH algorithm in the semantics.

The second contribution is that we give a set of syntactic conditions for a group of peers with nested *scope*s (the term we used for CA) to form a system with coherent behavior, and prove the sufficiency of the conditions based on the operational semantics.

However, the syntactic conditions are not convenient enough for developers. To check a composed system, we must compare all its peers pairwisely. As the most important contribution of the paper, we propose a global-to-local design approach to facilitate system designers, inspired by the idea of Web service choreography.

Our global-to-local approach suggests a three steps development: write a global specification, validate it, and finally project it into a set of peer specification. We define a Global Protocol Language (GPL) for the description of the global protocols, and a projection algorithm for automatic generating the specification for each peer. The approach is superior in several aspects: the global specification is easier to write; the conditions for well-behaved systems become much simpler on the global level. The automatically generated peers, putting together, will show the intended behavior described by the protocol. The system they composed will never deadlock w.r.t. CEH.

We implement the CEH framework formally defined here. A Java demonstration will be discussed in the paper with some primitive recognition.

The rest of the paper is organized as follows. Section 2 devotes to language PPL and some relative concepts. The CEH algorithm is introduced informally in Section 3, and the conditions and properties for consistent systems are given in Section 4. The protocol language with a projection algorithm is given in Section 5. Then we formalize PPL and CEH in Section 6, with proofs for the sufficiency of conditions in Section 7. As a demonstration of CEH, we show an implementation of CEH for threads in Java in Section 8. Finally, related work and conclusion are given.

2 A Peer Language

In order to formally study the cooperative systems with CEH, we define the Peer Process Language (PPL) to describe individual peers in the first. There have been some efforts to design such a language. COordinated Atomic LAnguage (COALA) is aimed to design CA actions [18].

$$
\begin{array}{lcll}
A & ::= & \mathsf{skip} & \text{(skip)} \\
 & | & BA & \text{(basics)} \\
 & | & [e_1, \ldots, e_l]BA & \text{(ex-decl.)} \\
 & | & A; A & \text{(sequential)} \\
 & | & sn : \{R, A, EH\} & \text{(scope)} \\
EH & ::= & \overline{e : A} & \\
BA & ::= & a \quad \text{(local act.)} & \\
 & | & c! \quad \text{(send)} \quad | \quad c? & \text{(receive)}
\end{array}
$$

Figure 1: Syntax of PPL

It is a formal language containing features such as pre/post conditions. Issarny [11] proposed an object-oriented language supporting CEH. The language is rich with many details, e.g., arithmetic expression and procedure definition. For easily understanding and formal proving, we define a small language as an extension of CSP [10], to capture the kernel of a peer with respect to CEH.

The language PPL is built on three fundamental concepts.

- Peer. A distributed system consists of several peers (independent subsystems) which communicate with one another.
- Exception interface. Each activity performed by a peer may either complete successfully, or fail and cause an exception. So each activity should be annotated by all the potential exceptions it may cause.
- Exception block. Programs are structured as blocks to confine exception [8]. Just like EH in common programming languages, an exception block contains a normal activity and a group of exception handlers. In addition, each block contains a list of names of the peers involved.

The syntax of PPL is given in Figure 1. Here A denotes an activity. BA is the basic activity which is either a local one or a communication. A local activity abstracts some real computations done by a peer. We will use $a, a_1, \ldots$ to denote local activities, and use $c!$, $c?$ for the sending and receiving through channel c. The exception declaration $[e_1, \ldots, e_l]BA$ means that the execution of BA may fail and cause one of the declared exceptions. A scope is an exception block, which has a name sn and a body consisting of three components: R is a name-set of the participating peers; A is the normal activity and EH is exception handlers. Conceptually, EH is a finite function from exception names to recovery activities, if $EH = e_1 : A_1, ..., e_l : A_l$, then $EH(e_i) = A_i$, $(i \in \{1, ..., l\})$. For scope sc, we use $sc.\mathsf{Name}$, $sc.\mathsf{R}$, $sc.\mathsf{A}$ and $sc.\mathsf{EH}$ to denote its name, set of participating peers' names, normal activity and exception handlers respectively.

Definition 1. *A* peer *is defined as a pair, (r, sc), where r is name and sc is behavior. The behavior is a scope. We will use α, α_1 for peer definitions, and r, r^1 for peer names. We will use $\alpha.\mathsf{Name}$ and $\alpha.\mathsf{SC}$ to denote name and behavior of α.*

Definition 2. *A* Composed System *(simply, a* system*) consists of several peers. We will denoted it as $\{\alpha_1, \ldots, \alpha_n\}$.*

By scanning a system, we can build up a *context*

$$\Gamma : \mathsf{R} \rightarrow \mathsf{SNames} \rightarrow \mathsf{Scopes}$$

Here R is the set of peer names in the system, SNames the set of scope names, and Scopes the set of scopes. $\Gamma(r)(sn)$ is the definition of scope named sn in peer r.

3 CEH Algorithm: Informal Description

This section roughly explains how Randell's CEH algorithm [22] works. We will formalize it by operational semantics in Section 6.

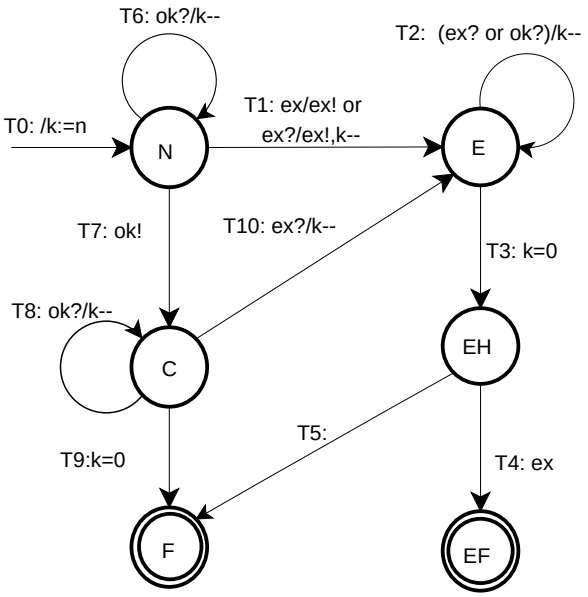


Figure 2: State Graph

In a distributed system, each peer will execute a series of activities encapsulated in a hierarchy of scopes. In the execution of a scope, a peer may be in one of six states, as depicted in Figure 2, where N is the normal state. To facilitate a two-step commitment, C and F are used to denote complete and finish states respectively. Other states are special for CEH, where E means the peer or its partners encounter some exceptions, EH means the peer is handling an exception, and EF means that the peer fails to handle an exception.

Figure 2 is the state transition graph of a peer in a scope with respect to CEH. Here `T1`, `T2`, ... are labels of transitions, `k` is a counter. The transitions are marked with "*event*/*activity*" pairs. The event `ex` indicates the peer encounters an exception. Moreover, `ex?` and `ex!` stand for receiving and sending an exception notification, `ok?` and `ok!` stand for receiving and sending a complete notification, respectively. For example, `T6:ok?/k--` means when receiving a complete notification, the peer decreases `k` and does the transition.

Initially, a peer begins a scope at state N and sets the counter `k` as the number of partners in the scope (`T0`). If an exception occurs or an exception notification is received, the peer stops its normal flow, jumps to E and informs all partners (`T1`). To work collaboratively, the peer can not start its handling before it knows that all peers in the same scope have entered state E (thus, $\texttt{k} = 0$). When the peer receives a notification from some partner, the counter decreases (`T2`). When `k` becomes zero, the peer calls an *exception resolution* algorithm to determine an exception to handle (`T3`). If an exception occurs in a handler, the scope is aborted and the exception propagates to the outer scope (`T4` to EF). Otherwise, if the handling ends normally, the peer exits the scope (`T5`).

On the other hand, if a peer completes the execution in a scope, it employs a two-step commitment. Firstly, it sends a complete notification to all partners and turns to state C (`T7`). Then, it waits there for the messages from its partners. If any partner encounters an exception, the peer jumps to E (`T10`) and acts as what is stated above. Otherwise, each partner will send a complete notification to the peer (thus, $k = 0$), it turns to F and finishes the scope (`T9`).

In transition `T3`, we need an algorithm for the peers to determine one exception to handle when some exceptions happened. Many strategies can be adopted here, e.g., [4] organized all exceptions into a tree and handled the minimal common ancestor when multiple exceptions detected, [20] suggested a complete order for peers. The lexical order of handlers in a scope can also be used here. Other considerations are possible. In the formal definition in Section 6, we wrap the strategy into a function and leave it to the implementation.

In the previous paragraphs, we describe how a peer acts in a single scope. Generally, during the

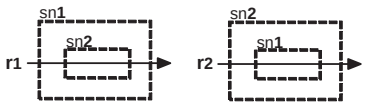


Figure 3: Inconsistent Scopes

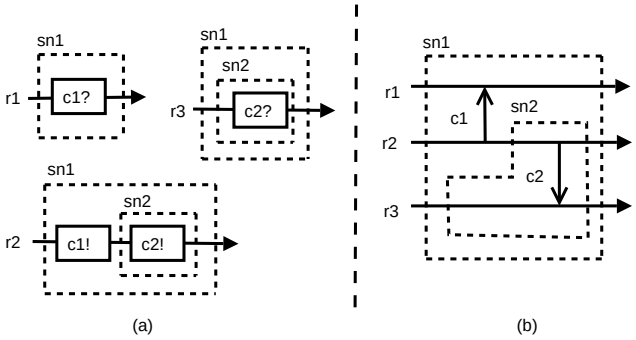


Figure 4: Consistent Scopes

execution, a peer will go through a set of nested scopes. Before a peer enters a nested scope, it will push the all the information of current scope, including its state, the uncompleted activities and the counter k, into a stack. Once the peer successfully finishes the nested scope, it pops the stack and resumes the execution of this scope. When executing a nested scope, the peer may encounter an exception while handling another exception; also it may receive an exception notification from a partner in an outer scope. In both situations, the peer will abort the nested scope and handle the exception in an outer one.

We will formalize the algorithm by detailed transition rules in Section 6.

4 Correctness and Consistent Conditions

Now we consider the correctness of the algorithm. Intuitively, for a composed system, we want all peers to be coherent in both normal and exception situations. Basically, the following requirements should be satisfied.

REQ-1. If no peer encounters an exception, all scopes will be completed normally.

REQ-2. If a peer encounters an exception in a scope, all peers involved in the same scope will stop normal execution and go to handle a same exception.

REQ-3. A system is deadlock-free with respect to the CEH procedures.

These requirements seem simple, but not any set of peers can form such a coherent system. For example, the two peers depicted in Figure 3 can not work cooperatively. Even if the two peers are lucky to avoid all exceptions and complete all normal activities of the inner scopes. Then, $r1$ will notify $r2$ and wait for $r2$ to complete sn_2. At the same time, $r2$ will wait for $r1$ to complete sn_1, and thus they fall into deadlock.

4.1 System Consistency

The counter-example in Figure 3 shows some clue for a consistent system: all peers in the system should have inter-compatible scope hierarchies. Figure 4 (a) shows a positive example,

$$
\mathsf{isCo}(P,Q) = \begin{cases} true & \text{if } P = skip \vee P = \epsilon \\ \mathsf{isCo}(P',Q) & \text{if } P = ba; P',\ ba \text{ is a basic activity} \\ \mathsf{isCo}(P',Q) & \text{if } P = sc; P' \wedge sc.\mathsf{Name} \notin \mathsf{SN}(Q) \\ \mathsf{isCo}(P',Q') & \text{if } P = sc_1; P' \wedge Q = sc_2; Q' \\ & \quad \wedge sc_1 \text{ is scope-consistent with } sc_2 \\ false & \text{if } P = sc_1; P' \wedge Q = sc_2; Q' \\ & \quad \wedge sc_1.\mathsf{Name} \in \mathsf{SN}(Q) \wedge sc_2.\mathsf{Name} \in \mathsf{SN}(P) \\ & \quad \wedge sc_1 \text{ is not scope-consistent with } sc_2 \\ \mathsf{isCo}(Q,P) & \text{otherwise} \end{cases}
$$

Figure 5: Consistent Activity

where all peers can be merged into a global graph, depicted as Figure 4 (b), where neither any two scopes overlap, nor any message goes across the boundary of scopes. In the following of this subsection, we will propose some syntactical conditions to capture these intuitions. To focus on CEH, we assume the execution of normal sending, receiving and local activities will always terminate, either normally or exceptionally. Then the conditions can be given as a definition for *consistent system*.

Definition 3. *We say system* $\{\alpha_1, \ldots, \alpha_n\}$ *is* consistent*, if the following conditions hold.*

- *Any two scopes appearing in same peer must have different names.*
- *if one scope* $(sn_1 : \{R_1, A_1, EH_1\})$ *is nested in the other* $(sn_2 : \{R_2, A_2, EH_2\})$*, i.e., it is in* A_2 *or* EH_2*, then* $R_1 \subseteq R_2$*.*
- *handlers in the outmost scope for each peer intend to perform a last-wish recovery, i.e., no exception will be thrown any more.*
- *For any* $\alpha_i, \alpha_j \in \{\alpha_1, \ldots, \alpha_n\}$*,* $\alpha_i.\mathsf{SC}$ *is "scope-consistent" with* $\alpha_j.\mathsf{SC}$*. The concept "scope-consistent" is defined below.*

Definition 4 (Scope-Consistent)**.** *Scopes* sc_1 *and* sc_2 *are consistent, if* $sc_1.\mathsf{Name} = sc_2.\mathsf{Name} \wedge sc_1.\mathsf{R} = sc_2.\mathsf{R}$*, furthermore,* $sc_1.\mathsf{A}$ *and* $sc_2.\mathsf{A}$*,* $sc_1.\mathsf{EH}$ *and* $sc_2.\mathsf{EH}$ *are consistent respectively.*

Exception handers eh_1 *and* eh_2 *are consistent, if* $\mathsf{dom}\, eh_1 = \mathsf{dom}\, eh_2$*, and* $eh_1(e)$ *is consistent with* $eh_2(e)$ *for each* $e \in \mathsf{dom}\, eh_1$*.*

Activities P *and* Q *are consistent, if each pair of scopes with the same name appearing both in* P *and* Q *are consistent, and these scopes have the same relative order and nest structures.* □

In other words, two activities P and Q are scope-consistent, if $\mathsf{isCo}(P,Q) = true$, where function $\mathsf{isCo}()$ is defined in Figure 5, and $\mathsf{SN}(P)$ denotes the set of scope names appeared in P.

The system shown in Figure 4 (a) is consistent, which can be merged into a global picture (b) without intersectant scopes. In PPL, the peers are:

$$
\begin{aligned}
\alpha_1 &= (r^1, sn_1 : \{R_1, c1?, \epsilon\}) \\
\alpha_2 &= (r^2, sn_1 : \{R_1, c1!; sn_2 : \{R_2, c2!, \epsilon\}, \epsilon\}) \\
\alpha_3 &= (r^3, sn_1 : \{R_1, sn_2 : \{R_2, c2?, \epsilon\}, \epsilon\})
\end{aligned}
$$

where $R_1 = \{r^1, r^2, r^3\}, R_2 = \{r^2, r^3\}$.

4.2 Properties of consistent systems

In this subsection, we propose several propositions and theories to show some properties of consistent systems.

We say a peer r terminates scope sc if it is either in scope sc and its state is F, or it has exited from sc. We say a scope sc terminates if all its participating peers terminate the scope.

For the consistent systems, we propose several propositions and a theorem here.

Proposition 1. ***corresponding [REQ-1]*** *In a consistent system, supposing all relative peers have entered scope sn, if no peer will encounter any exception, then scope sn will terminate.*

Proposition 2. ***corresponding [REQ-2]*** *Suppose one or more exceptions happen in scope sn, if all relative peers have entered sn and no exception happens in outer scopes, these peers will enter state* E*, and determine an identical exception to handle.*

Proposition 3. *Suppose all relative peers have entered sn, and all its inner scopes will terminate, if some exceptions happen, either in sn, or its outer scopes, or in inner scopes and propagate to sn, then sn will terminate.*

Theorem 1. ***corresponding [REQ-3]*** *A* consistent *system will never deadlock.*

These propositions and theorem will be formally proved in Section 7 based on an operational semantics of PPL defined there.

Having the properties, we can determine whether a system is coherency by checking its peers according to the conditions given in previous subsection. For these complicated conditions, the checking would not be an easy job. To make the system developing easier, we will present a design approach for consistent systems in the next section. The design approach enables us to specify the system as a whole, and then generate the specification (with all scope structures) of each peer automatically.

5 Global-to-Local: Design and Implementation

It is not easy to develop a set of peers to form a consistent system, due to the complex consistent conditions (Section 4). The situation is even worse if the peers are developed by independent organizations for business applications. To overcome this difficulty, we propose a global-to-local development methodology, where a protocol is specified from a global viewpoint, then is used to generate the peers which will always make up a consistent system. For this, we define in section a protocol language GPL (for Global Protocol Language), then propose a projection to generate peers in PPL from GPL specifications. The validation conditions are also discussed.

5.1 Protocol Language

GPL is designed for writing global scenario (*protocols*) of several peers by specifying their collaborative observable behavior in both normal execution and exception handling. GPL allows sub-protocols to be nested in any depth:

$$
\begin{array}{lcll}
C & ::= & \langle cn, R, A, EH\rangle & \\
EH & ::= & \overline{e:A} & \\
BA & ::= & a^i \quad | & c^{[i,j]} \quad i \neq j \\
A & ::= & BA \quad | & [e_1, \ldots, e_l]BA \\
 & | & C \quad | & A; A
\end{array}
$$

A GPL protocol takes a similar form as a activity in PPL, except communications.

A protocol C consists of a name cn, a peer-name-set R, an activity A and some exception handlers EH. A basic activity BA is either a local one a^i, where i indicates that its performer is peer r^i, or a communication $c^{[i,j]}$ from r^i to r^j. An activity A can be a basic, perhaps with exception declaration, a sub-protocol, or a sequential composition. The exception declaration is similar to that in PPL, while $c^{[i,j]}$ with exception declaration means the exceptions may be raised in r^i or r^j during the communication.

Similar to PPL, not every protocol following GPL syntax makes sense. Here the well-formedness condition is much simple:

$$
\begin{array}{rcl}
\pi(a^i, j) & \widehat{=} & \begin{cases} a^i & \text{if } j = i \\ \mathsf{skip} & \text{if } j \neq i \end{cases} \\
\pi(c^{[i,j]}, k) & \widehat{=} & \begin{cases} c! & \text{if } k = i \\ c? & \text{if } k = j \\ \mathsf{skip} & \text{otherwise} \end{cases} \\
\pi(A_1; A_2, i) & \widehat{=} & \pi(A_1, i); \pi(A_2, i) \\
\pi([e_1, \ldots, e_l]B, i) & \widehat{=} & \overline{[e_1, \ldots, e_l]}\pi(B, i) \\
\pi(\overline{e : A}, i) & \widehat{=} & \overline{e : \pi(A, i)} \\
\pi(\langle cn, R, A, EH \rangle, i) & \widehat{=} & \begin{cases} cn{:}\{R, \pi(A, i), \pi(EH, i)\} & \text{if } i \in R \\ \mathsf{skip} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 6: *Natural Projection*

Definition 5. *A protocol is well-formed, if the protocol and all sub-protocols have different names, each performer of the activities and handlers of a protocol are declared in its peer-name-set, and each protocol's peer-name-set is a subset of its containing protocol's.*

5.2 Projection and Implementation

A protocol describes collective behaviors of a group of peers from a global viewpoint. However, protocols are not executable, their behavior should be implemented by cooperative peers. In this subsection, we propose a projection algorithm for extracting the behavior description of each peer from global protocols. Figure 6 defines a simple projection, which partitions a protocol to produce the designated peer following its structure. We name it the *natural projection*.

With the projection, we can generate a composed system from a protocol. Suppose C is a protocol with n peers ($|C.\mathsf{R}| = n$). We define

$$\pi(C) \ \widehat{=} \ \{(C.\mathsf{R}(i), \pi(C, i)) \mid i \in 1..n\}$$

where $C.\mathsf{R}(i)$ is the name of the ith peer in C. We call $\pi(C)$ the composed system defined by C.

For a well-formed protocol C, system $\pi(C)$ satisfies all consistent conditions given in Section 4. Thus we have:

Theorem 2. *If C is a well-formed protocol, $\pi(C)$ is a consistent composed system.*

Corollary 1. *If C is a well-formed protocol, $\pi(C)$ is deadlock free w.r.t. CEH.*

Further connection between PPL and GPL can be investigated if semantics for both languages are given. Under some conditions, C and $\pi(C)$ are behavioral equal in some sense. Interested reader can refer to our previous work [13].

Now we have a global-to-local approach to design composed systems with CEH facility, which can be summarized as follow steps:

1. Identify peers taking part in the work, and specify their collective behavior, either in normal case or in the exception case from a global view point with GPL.
2. Check whether the global specification is a well-formed protocol.
3. Project the global specification to a set of peer specifications, and fill in the details for the local activities.

Then we have a consistent composed system with collaborated exception handling support.

For a solid theoretical foundation of this methodology, we should give formal semantics for the PPL and GPL and prove some properties, including the propositions and theories given in last section. This is the work presented below.

[**Basic Cases**]

$r:\langle \sigma, sn, s, \mathsf{skip}, \epsilon, \theta\rangle \longrightarrow r:\langle \sigma, sn, s, \epsilon, \epsilon, \theta\rangle$

$r:\langle \sigma, sn, s, a, \epsilon, \theta\rangle \xrightarrow{a} r:\langle \sigma, sn, s, \epsilon, \epsilon, \theta\rangle$

$r:\langle \sigma, sn, s, [e_1, \ldots, e_l]A, \epsilon, \theta\rangle \longrightarrow r:\langle \sigma, sn, s, A, \epsilon, \theta\rangle$

[$\mathsf{N} \to \mathsf{E}$]

$r:\langle \sigma, sn, \mathsf{N}, [e_1, \ldots, e_l]A, \epsilon, \theta\rangle \longrightarrow r:\langle \sigma, sn, \mathsf{E}, \mathit{notify}(e_i), \epsilon, \theta\rangle \quad i \in 1..l$

$r:\langle \sigma, sn, \mathsf{N}, A, (e, sn, r')^\frown \omega, \theta\rangle \longrightarrow r:\langle \sigma, sn, \mathsf{E}, \mathit{notify}(e), \omega, \theta \oplus \{r' \mapsto e\}\rangle$

[$* \to *$]

$r:\langle \sigma, sn, s, A, (\mathsf{C}, sn, r')^\frown \omega, \theta\rangle \longrightarrow r:\langle \sigma, sn, s, A, \omega, \theta \oplus \{r' \mapsto \mathsf{C}\}\rangle$

[$\mathsf{N} \to \mathsf{C}$]

$r:\langle \sigma, sn, \mathsf{N}, \epsilon, \epsilon, \theta\rangle \longrightarrow r:\langle \sigma, sn, \mathsf{N}, \mathit{notify}(\mathsf{C}), \epsilon, \theta\rangle$

[$\mathsf{E}, \mathsf{C} \to \mathsf{E}$]

$$\frac{s = \mathsf{E} \vee s = \mathsf{C}}{r:\langle \sigma, sn, s, A, (e, sn, r')^\frown \omega, \theta\rangle \longrightarrow r:\langle \sigma, sn, \mathsf{E}, A, \omega, \theta \oplus \{r' \mapsto e\}\rangle}$$

[$\mathsf{E} \to \mathsf{EH}$]

$$\frac{\mathsf{N} \notin \mathsf{ran}\,\theta}{r:\langle \sigma, sn, \mathsf{E}, \epsilon, \omega, \theta\rangle \longrightarrow r:\langle \sigma, sn, \mathsf{EH}, \mathit{handler}_\Gamma(r, sn, \theta), \omega, \mathsf{dom}\,\theta \times \{\mathsf{N}\}\rangle}$$

[$\mathsf{C}, \mathsf{EH} \to \mathsf{F}$]

$$\frac{s = \mathsf{EH} \vee (s = \mathsf{C} \wedge \mathsf{ran}\,\theta = \{\mathsf{C}\})}{r:\langle \sigma, sn, s, \epsilon, \epsilon, \theta\rangle \longrightarrow r:\langle \sigma, sn, \mathsf{F}, \epsilon, \epsilon, \theta\rangle}$$

[**Scope switch**]

$r:\langle \sigma, sn, s, sn':\{R', A', EH'\}; A, \epsilon, \theta\rangle \longrightarrow r:\langle sn:(s, A, \theta)^\frown \sigma, sn', \mathsf{N}, A', \epsilon, R'\times\{\mathsf{N}\}\rangle$

$r:\langle sn':(s', A', \theta')^\frown \sigma, sn, \mathsf{F}, \epsilon, \epsilon, \theta\rangle \longrightarrow r:\langle \sigma, sn', s', A', \epsilon, \theta'\rangle$

$r:\langle sn':(s', A', \theta')^\frown \sigma, sn, \mathsf{EH}, [e_1, \ldots, e_l]A, \epsilon, \theta\rangle \longrightarrow r:\langle \sigma, sn', \mathsf{E}, \mathit{notify}(e_i), \epsilon, \theta'\rangle \quad i \in 1..l$

[$* \to \mathsf{EF}$]

$$\frac{sn_0 \in \mathsf{dom}\,\sigma \;\vee\; sn_0 = sn'}{r:\langle sn':(s', A', \theta')^\frown \sigma, sn, s, A, (e, sn_0, r^i)^\frown \omega, \theta\rangle \longrightarrow r:\langle \sigma, sn', s', A', (e, sn_0, r^i)^\frown \omega', \theta'\rangle}$$

[**Sequential**]

$$\frac{r:\langle \sigma, sn, s_1, A_1, \omega, \theta\rangle \xrightarrow{t} r:\langle \sigma, sn, s_2, A_1', \omega', \theta'\rangle}{r:\langle \sigma, sn, s_1, A_1; A_2, \omega, \theta\rangle \xrightarrow{t} r:\langle \sigma, sn, s_2, A_1'; A_2, \omega', \theta'\rangle}$$

Figure 7: Local Transition Rules

6 Semantics of PPL with CEH

To prove the propositions and theorems listed above, we formalize the CEH algorithm by a formal operational semantics. Firstly, we define configuration of a single peer and a system, then give transition rules for configurations.

6.1 Configurations

A peer may enter a series of nested scopes. A configuration of a peer is a tuple:

$$r:\langle \sigma, sn, s, A, \omega, \theta\rangle$$

where r is the peer's name, sn is the name of scope that the peer is executing, s is a state as we have in Figure 2, A is the upcoming activity the peer will do. σ a stack recording the states of all scopes where the peer has entered but not exited yet except current one. Message queue ω has elements of the form (m, sn', r'), which means peer r' sends message m from scope sn'. Table θ records the states of all partners in scope sn, which is a mapping $\Gamma(r)(sn).\mathsf{R} \to \{\mathsf{N}, \mathsf{C}\} \cup \{e \mid \text{e is an exception}\}$. We will use ϵ to represent something (stack, queue, or set) empty.

For a peer defined as $(r, sn:\{R, A, EH\})$, its initial configuration is:

$$r:\langle \epsilon, sn, \mathsf{N}, A, \epsilon, R \times \{\mathsf{N}\}\rangle$$

This means that the peer is in the normal state initially (represented by N), with its activity A, and all its partners in this scope are in normal state (represented by map $R \times \{\mathsf{N}\}$).

The global configuration of a system consists of configurations of all its peers:

$$G = \{f_1, \ldots, f_n\}$$

It is the initial configuration if each f_i is the initial configuration of peer r^i.

6.2 Local Transition Rules

Transition rules for a peer locally are given in Figure 7. A rule is in the form of $f \xrightarrow{t} f'$, where f and f' are configurations, t an observable event. When t is null, we simply omit it. The rules are classified into several groups. Here are explanations:

[**Local**]

$$\frac{f_i \xrightarrow{t} f_i' \quad i \in 1..n}{\{\ldots, f_i, \ldots\} \xrightarrow{t} \{\ldots, f_i', \ldots\}}$$

[**Communication**]

$$\frac{\exists i, j \in 1..n \bullet \quad f_i = r^i : \langle \sigma^i, sn, s^i, c!; A^i, \epsilon, \theta^i \rangle \text{ and } f_j = r^j : \langle \sigma^j, sn, s^j, c?; A^j, \epsilon, \theta^j \rangle}{\{\ldots, f_i, \ldots, f_j, \ldots\} \xrightarrow{c} \{\ldots, r^i : \langle \sigma^i, sn, s^i, A^i, \epsilon, \theta^i \rangle, \ldots, r^j : \langle \sigma^j, sn, s^j, A^j, \epsilon, \theta^j \rangle, \ldots\}}$$

[**Complete notification**]

$$\frac{sn_i = sn_j}{\begin{array}{l}\{\ldots, r^i : \langle \sigma_i, sn_i, s_i, A_i, \omega_i, \theta_i \rangle, \ldots, r^j : \langle \sigma_j, sn_j, \mathsf{N}, send(r^i, \mathsf{C}); A_j, \omega_j, \theta_j \rangle \ldots\} \longrightarrow \\ \quad \{\ldots, r^i : \langle \sigma_i, sn_i, s_i, A_i, \omega_i \frown (\mathsf{C}, sn_j, r^j), \theta_i \rangle, \ldots, r^j : \langle \sigma_j, sn_j, \mathsf{C}, A_j, \omega_j, \theta_j \oplus \{r^j \mapsto \mathsf{C}\} \rangle \ldots\}\end{array}}$$

[**Exception notification**]

$$\frac{sn_j \in \mathsf{dom}\, \sigma_i \vee sn_j = sn_i}{\begin{array}{l}\{\ldots, r^i : \langle \sigma_i, sn_i, s_i, A_i, \omega_i, \theta_i \rangle, \ldots, r^j : \langle \sigma_j, sn_j, E, send(r^i, e); A_j, \omega_j, \theta_j \rangle \ldots\} \longrightarrow \\ \quad \{\ldots, r^i : \langle \sigma_i, sn_i, s_i, A_i, \omega_i \frown (e, sn^j, r^j), \theta_i \rangle, \ldots, r^j : \langle \sigma_j, sn_j, E, A_j, \omega_j, \theta_j \oplus \{r^j \mapsto e\} \rangle \ldots\}\end{array}}$$

Figure 8: Global Transition Rules

[**Basic Cases**] A peer can execute a local activity when its message queue is empty. The execution may success or cause an exception if it has declared any one. If an exception appears, the peer will enter state E, as the the rule below.

[$\mathsf{N} \rightarrow \mathsf{E}$] A peer may cause an exception if the executed activity declares some exception, and also it may receives an exception notification. Then, it will turn to E, record its partner's state, and notify its partners by *notify*. The rule of *notify* is described in the relative global rule below. Here $\oplus$ denotes overriding.

[$* \rightarrow *$] No matter in which state, when receiving a completion notification, a peer records it.

[$\mathsf{N} \rightarrow \mathsf{C}$] When finishing normal execution, the peer notifies its partners by *notify*(C).

[$\mathsf{E}, \mathsf{C} \rightarrow \mathsf{E}$] When receiving an exception notification in state E/C, the peer updates its state table and turns to E.

[$\mathsf{E} \rightarrow \mathsf{EH}$] When a peer is in state E, and all its partners have exited normal state, thus, the state table contains no "N", the peer begins to handle exception. Here $handler_\Gamma$ is a function taking a peer name r, a scope name sn, and a state table θ as parameters. It determines an exception and finds the handler for it. We wrap the resolution strategy into a function *resolve* and leave it to the practice.

$$handler_\Gamma(r, sn, \theta) \mathrel{\widehat{=}} \Gamma(r)(sn).\mathsf{EH}(resolve(\theta))$$

[$\mathsf{C}, \mathsf{EH} \rightarrow \mathsf{F}$] If a peer knows all partners of its current scope complete, or it finishes exception handling without any fault, it can exit current scope safely (ref. rule [**Scope switch**]).

[**Scope Switch**] If a peer meets a new scope, or finishes its work in a scope, it switches in/out the scope. Here $R \times \{\mathsf{N}\}$ means that all peers are in state N. If an exception happens or is re-thrown in a handler, it is propagated to the enclosing scope.

[$* \rightarrow \mathsf{EF}$] If an exception notification from outer scope arrives, the peer aborts immediately current scope, deletes messages of current scope. The premise means sn_0 is an outer scope of r's current scope (named sn). Here ω' is the message queue obtained by filtering out all messages of form $(\text{-}, sn, \text{-})$ from ω.

6.3 Global Transitions

The global transition rules are listed in Figure 8, where we use "..." to denote some unchanged part. A local transition induces a global transition ([**Local**]). Two peers communicate with each other only when both are in the same scope and ready to communicate via the same channel ([**Communication**]). The communication is synchronous which is similar as in CSP [10].

Some local rules transit to a configuration with a call $notify(m)$ to send m to all partners in the same scope, where m is C or some e. Without careful management, asynchronous approach may induce deadlocks. A scenario is given in our report [3]. Here we realize $notify(m)$ by a group of synchronous sending, where $send(r^i, m)$ sends m to r^i. An invocation $notify(m)$ in scope sn by r is equal to "$send(r^1, m); \ldots; send(r^k, m)$", where $\{r^1, \ldots, r^k\} = \Gamma(r)(sn).\mathsf{R} - \{r\}$. It was better if we had parallel construction to implement it as "$send(r^1, m) \parallel \ldots \parallel send(r^k, m)$".

In rule [**Complete notification**], peer r^j sends a complete message to partners r^i in sn_j that is recorded in message queue. The premise asks the sender and receiver in the same scope. Rule [**Exception notification**] is similar, except the requirement for r^i is looser. Here $sn_j \in \mathsf{dom}\,\sigma_i$ means that r^i has entered the scope sn_j but not left.

7 Proofs for Theorem 1

Properties of consistent systems have been listed in Subsection 4.2. Now we given the skeleton of the proof for Theorem 1 based on the formalization above. Firstly, some lemmas.

Lemma 1. *A peer can exit a scope if one of the following situations occurs:*
1. Its state table is full of C*;*
2. Some exceptions occurred in some outer scopes;
3. The peer finishes its exception handling, or encounters an exception when handling exception.

By Lemma 1, we can prove Theorem 1 in two steps, in the first, prove all peers can cooperatively exit a scope in each case of Lemma 1, and then apply structure induction on hierarchies of scopes.

Lemma 2. *If peer r participates in scopes sn_1 and sn_2, and sn_2 is in sn_1, then for any r's configuration $r : \langle \sigma, sn, s, A, \omega, \theta \rangle$, if $sn = sn_2$, then, $sn_1 \in \mathsf{dom}(\sigma)$.*

We define a partial relation *preceding* for scopes.

Definition 6. *We say sn_1 is preceding of sn_2, if there exists a peer r with $sc_1 = \Gamma(r)(sn_1)$, $sc_2 = \Gamma(r)(sn_2)$, and sc_1 appears before sc_2 syntactically.*

Lemma 3. *If all relative peers have entered scope sc, an attempt to notify(C) or notify(e) in sc will always succeed if no exception happen in outer scopes.*

Lemma 4 (Deadlock-Free)**.** *A peer can always consume its message queue.*

Lemma 5. *If all peers have entered their exception handlers for a same exception e in scope sn, suppose all inner scopes can always terminate and no exception will happen in outer scopes, they will finish recovery and terminate sn.*

Lemma 6. *If scopes $sn_1, ..., sn_l$ have a common direct enclosing scope, then they can be sorted to a sequence $sn'_1, ..., sn'_l$ such that for any $1 \le i < j \le l$, sn'_j is not* preceding *of sn'_i, and sn'_j must appear in $\Gamma(r)(sn'_i).\mathsf{A}$ for some r if sn'_i appears in $\Gamma(r)(sn'_j).\mathsf{A}$, i.e., all scopes in exception handlers are sorted after those in normal activities.*

Lemma 7. *For a scope sn' and its direct-enclosed scope sn, suppose all peers related to sn have entered sn', all scopes* preceding *sn are finished, and one peer has entered sn, then the rest peers will enter sn eventually, unless exceptions happen in some enclosing scopes (include sn').*

Theorem 3. *In a consistent system, suppose all relative peers enter a scope, this scope will always terminate.*

From all the lemmas and propositions given above, we can easily have Theorem 1.

Proof for **Theorem 1**. For a *consistent* system $\{\alpha_1, ..., \alpha_n\}$, we know $\alpha_j.\mathsf{SC.R} = \{\alpha_i.\mathsf{Name} \mid i \in \{1, .., n\}\}$, suppose $\alpha_j.\mathsf{SC.Name}$ is sn, the consistent condition ensures that all peers required by sn will enter it. By Theorem 3, the system will always terminate. □

Now we finish the work for building the theoretical foundation for our design methodology.

8 Experiment

The operational semantics given in Section 6 is easy to implement. As a demonstration, we developed a Java framework where threads can handle exceptions cooperatively.

We define classes GlobalException and Peer, where Peer extends java.lang.Thread, which has attributes for the stack, state table, message queue, and methods such as enterScope(String name), waitToExitScope(), inforAllException(GlobalException ge), inforAllComplete(). Method run() implements a hierarchy of scopes, in which each scope is a try-catch block with special form, where the first statement is always a method call of enterScope(), and the last is waitToExitScope(). Furthermore, non-global exceptions must be handled in the scope, but global exceptions are thrown to outer scopes.

The implementation can be revised for distributed applications, where several methods should be promoted to communicate with remote objects. Also, the CEH mechanism can be implemented on the JVM level to write eleganter codes, by applying the approach proposed in paper [23] which revises the exception mechanism for the new feature "Future" in Java 5.0.

9 Related work

Exception handling (EH) has become indispensable ingredient in programming languages and system development fields incrementally after Goodenough's seminal work on structural EH [8]. Randell presented in [14] the rationale behind a method for structuring complex computing systems by the use of "recovery blocks" and "conversations". In [4], Campbell and Randell proposed techniques for structuring forward error recovery method in asynchronous systems. In [15] , Romanovsky developed a new atomic action scheme that did not impose any peer synchronization on action exit. Xu *et al.* [20, 19] presented a scheme for coordinated error recovery between multiple interacting objects in concurrent OO systems, and developed a conceptual model and algorithm especially for distributed object systems. However, all these works are presented informally. Neither the well-formedness condition nor semantics is presented.

There have been some effort to design a language to support the concept of CA. COALA [18] is aimed to design CA actions. It is a formal language containing features such as pre/post conditions. Issarny [1] proposed a language ϵ_{CSP} to support EH in concurrent systems. A framework for EH in parallel OO language is proposed in [11], where a system is a group of recoverable actions. Each action involves several peers and a special "coordinator" works as a control center to decide whether the action fails and directs the others to handle an exception. Issarny [11] proposed an OO language supporting CEH. The language is rich with many details, i.e. arithmetic expression and procedure calling. However, these models are unsuitable for the decentralized systems, because it is hard to find a reliable peer with overwhelming power. A survey about exception handling models developed for concurrent object systems can be found in [16].

Current research trends in EH is outlined in [12], including EH in human-centered systems such as workflow. Hagen and Alonso [9] presented a solution for implementing reliable workflow processes by using EH. Romanovsky *et al.* [17] proposed coordinated forward error recovery for composite Web services, and defined the notion of Web Services Composition Action (WSCA), which allows constructing composite Web services in terms of dependable actions.

A framework to verify CEH is presented in [7], where a system is modeled by specification language like Alloy and B, and programmers can write and check properties using a constraint solver. In the paper, three requirements commonly for systems are enumerated as examples. But the paper does not aim to give a sufficient condition for the well-behaved systems.

Model checking techniques have also been employed for verifying CEH systems. As the general case, model checking approaches typically suffer from the state space explosion problem, which is especially acute for large systems. An experiment [21] showed that the state space is too big to handle. A model of a manufacture controlling system involves huge state space, where the checking gives no result after one week of computation. The major advantage of our design approach is that it avoids this problem.

10 Conclusion

The coordinated exception handling (CEH) in distributed system is very complex due to the autonomous nature of the peers in the system. Current develop methods and languages almost consider from the perspective of a local peer. It is difficult to guarantee the composition of peers can cooperate properly. Based on this recognition, we present in this paper a global-to-local approach for building the composed system consisted of multiple independent peers which supports CEH. The contributions of this work can be summarized as follows:

- Propose a group of accurate syntactic conditions for a set of peers to form a well-behaved system.
- Present an efficient global-to-local design and implementation approach for cooperative systems with CEH.
- Formally define the CEH procedure by a set of operational rules, and formally proved these conditions are sufficient based on the operational semantics.

The PPL language used in our formalization can be seen as a try to introduce linguistic mechanism for specifying nested CA actions in distributed environment. With this formal framework, we have developed a demonstrative framework, which enable Java threads to handle exceptions cooperatively. We also have a tool to automatically generate skeletons of threads, with integrated CEH in each of the thread's code.

As a future work, it is meaningful to study the integration of CEH into existing Web services languages such as WS-BPEL and WS-CDL. Actually, Carbone *et al.* [6] has recognized that exception are indispensable for managing many real application situations. They identified that one missing thing in WS-CDL would be the ability of handling exceptions locally, with a standard local scoping rule. Therefore, it deserves further effort based on current formalization of CEH.

References

[1] Jean-Pierre Banâtre and Valérie Issarny. Exception handling in communication sequential processes. Technical Report 660, INRIA-Rennes, IRISA, 1992.

[2] D. M. Beder, A. Romanovsky, B. Randell, C. R. Snow, and R. J. Stroud. An application of fault tolerance patterns and coordinated atomic actions to a problem in railway scheduling. *SIGOPS Oper. Syst. Rev.*, 34(4):21–31, October 2000.

[3] Chao Cai, Zongyan Qiu, Hongli Yang, and Xiangpeng Zhao. Coordinated exception handling in web service. Technical report, 2007. Available as Prepring 2007-23 of Institute of Mathematics, Peking University, at: http://www.math.pku.edu.cn:8000/en/preindex.php.

[4] Roy H. Campbell and Brian Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, August 1986.

[5] Alfredo Capozucca, Nicolas Guelfi, and Patrizio Pelliccione. The fault-tolerant insulin pump therapy. In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *RODIN Book*, volume 4157 of *Lecture Notes in Computer Science*, pages 59–79. Springer, 2006.

[6] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming. Technical report, To be published by W3C., 2006. Available at `http://www.dcs.qmul.ac.uk/~carbonem/cdlpaper/workingnote.pdf`.

[7] Fernando Castor Filho, Alexander Romanovsky, and Cecília M. F. Rubira. Verification of coordinated exception handling. In Hisham Haddad, editor, *SAC*, pages 680–685. ACM, 2006.

[8] John B. Goodenough. Exception handling: issues and a proposed notation. *Communications of ACM*, 18(12):683–696, December 1975.

[9] Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Trans. Softw. Eng.*, 26(10):943–958, 2000.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[11] Valérie Issarny. Concurrent exception handling. In *Proceedings of Advances in Exception Handling Techniques, LNCS 2022*, pages 111–127. Springer, 2001.

[12] Dewayne E. Perry, Alexander Romanovsky, and Anand Tripathi. Current trends in exception handling. *IEEE Trans. Softw. Eng.*, 26(9):817–819, 2000.

[13] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proc. of WWW 2007, Banff, Canada*. ACM, 2007.

[14] B. Randell. System structure for software fault tolerance. *IEEE Trans. on Soft. Eng.*, SE-1(2):220–232, 1975.

[15] Alexander Romanovsky. Looking ahead in atomic actions with exception handling. In *Reliable Distributed Systems, 2001. Proceedings. 20th IEEE Symposium on*, 2001.

[16] Alexander Romanovsky and Jörg Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In *Proceedings of Advances in Exception Handling Techniques, LNCS 2022*, pages 147–164. Springer, 2001.

[17] Ferda Tartanoglu, Valérie Issarny, Alexander B. Romanovsky, and Nicole Lévy. Coordinated forward error recovery for composite web services. In *Proc. of 22nd Symposium on Reliable Distributed Systems*, pages 167–176. IEEE Computer Society, 2003.

[18] Julie Vachon, Didier Buchs, Mathieu Buffo, Giovanna Di Marzo, Serugendo, Brian Randell, Sascha Romanovsky, Robert Stroud, and Jie Xu. Coala - a formal language for coordinated atomic actions. Technical report, third year report, ESPRIT Long Term Research Project 20072 on Design for Validation, 1998.

[19] J. Xu, A. B. Romanovsky, and B. Randell. Coordinated exception handling in distributed object systems: from model to system implementation. In *Proceedings of 18th Inter. Conf. on Distributed Computing Systems*, pages 12–21, 1998.

[20] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecília M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proc. of the 25th International Symp. on Fault-Tolerant Computing*, pages 499–509, 1995.

[21] Jie Xu, Brian Randell, Alexander B. Romanovsky, Robert J. Stroud, Avelino F. Zorzo, Ercument Canver, and Friedrich W. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Trans. Computers*, 51(2):164–179, 2002.

[22] Jie Xu, Alexander B. Romanovsky, and Brian Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Trans. on Parallel and Distributed Systems*, PDS-11(10):1019–1032, October 2000.

[23] Lingli Zhang, Chandra Krintz, and Priya Nagpurkar. Supporting exception handling for futures in java. In *PPPJ*, volume 272 of *ACM International Conference Proceeding Series*, pages 175–184. ACM, 2007.

[24] Avelino F. Zorzo, Alexander B. Romanovsky, Jie Xu, Brian Randell, Robert J. Stroud, and Ian Welch. Using coordinated atomic actions to design safety-critical systems: a production cell case study. *Softw., Pract. Exper.*, 29(8):677–697, 1999.

Translation and Optimization for a Core Calculus with Exceptions

Cristina David and Cristian Gherghina and Wei-Ngan Chin

Department of Computer Science, National University of Singapore
{cristina,cristian,chinwn}@comp.nus.edu.sg

Modern programming languages have many useful features to help the construction of software. Being meant for the development of applications, their main aim is to offer a high degree of flexibility and ease of use to the programmer. Consequently, they become complex and hard to analyse. For instance, one important feature is exception handling. This feature is used to handle unusual conditions that can lead to errors, unless remedial actions are suitably taken. However, exceptions may induce non-local control flows that could make programs harder to analyse statically. This worry is one reason why a number of past proposals (including our own work [5]) on calculi to facilitate formal reasoning [6,1] have mostly ignored exceptions, in the name of simplicity. Some recent proposals [4,3] have begun to consider a core language with exceptions by adding both a `throw e` construct and a simplified `try` e_1 `catch (c v)` e_2 construct from the Java language. However, this simplified feature was not able to *succinctly* handle more advanced features, such as `try-finally` nor `try-with-multiple-catches`. Another proposal [2] directly adds these advanced features in their core language, but this is done at the price of a more complex formalization.

Our proposal is to perform the analysis on an intermediate simplified core calculus, avoiding the complexity of the source language, but without restricting the flexibility expected by the programmers. The two crucial requirements for our calculus are to be easy to analyse (**1**), and to be expressive enough as to allow the translation of more complex language constructs into it (**2**). For achieving the first goal (**1**), we endow our core calculus with a *unified view* of the control flow, which spans two dimensions:

- Unifies both normal and abnormal control flows. Unlike past works which simply add together the features of normal and abnormal control flows, our change is more fundamental since we provide a pair of unified language constructs (elaborated later) that would work for all kinds of control flows.
- Unifies the static control flow, where the syntax of the program directly deter-

mines which parts of the program may be executed next, and the dynamic control flow, where run-time values and inputs of the program are required to decide what to execute next. This is achieved by translating the break, continue, return constructs (specific to the static control flow), and the try-catch and raise constructs (specific to the dynamic control flow), into a unified control flow mechanism under our calculus.

An unexpected benefit is that our core calculus with exceptions is *as small as* the corresponding core calculus without exceptions. Designing analyses and optimizations for the core calculus is therefore much simpler than it would be for the source language! With regard to the current work's second goal (**2**), we prove the expressivity of the core calculus by providing a set of rewrite rules for translating a medium-sized imperative source language into it.

We refer to our design as a *calculus* rather than a *language* since our intention is to support a broader range of formal reasoning activities, including analysis, language design, compilation, optimization and verification. The resulting core calculus will essentially contain a core language and a set of rules (including translation) that facilitate formal reasoning. Our goal is for a core calculus that is *syntactically minimal and expressively maximal*. We shall describe an application of our calculus, namely optimization. In order to prove the soundness of the optimization rules, we shall formalise the calculus by providing a big-step operational semantics.

References

[1] Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe - Probably. In *ECOOP*, pages 389–418, 1997.

[2] Sophia Drossopoulou and Tanya Valkevych. Java Exceptions Throw No Surprises. Technical report, March 2000.

[3] Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. An uncaught exception analysis for Java. *Journal of Systems and Software*, 72(1):59–69, 2004.

[4] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[5] H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, Nice, France, January 2007.

[6] Tobias Nipkow and David von Oheimb. Java_{light} is Type-Safe - Definitely. In *POPL*, pages 161–170, 1998.

Concern Based Approach to Generating SCR Requirement Specification: a Case Study*

Ying Jin[1] Jing Zhang[1] Weiping Hao[1] Pengfei Ma[1]
[1]*College of Computer Science and Technology, Jilin University, Changchun, P.R. China*
jinying@jlu.edu.cn

Extended Abstract

SCR is a mature and widely used document driven requirement method, which emphasizes creating strict and complete formal requirement documents during software requirement analysis to serve the whole life circle[1]. However, due to the big gap between textual requirement description and formal requirement document, how to generate well defined, concise and strict formal requirement document from large scale and complex textual requirement statements has become a big challenge in application of SCR method.

In order to address this problem, a concern based approach to generating SCR requirement specification from prose description has been proposed[2]**, where concerns are used to bridge the gap between textual requirement definition and formal SCR specification. Concerns and their relationship are identified and specified, which will be used to guide identifying and specifying different constructs in SCR requirement documents.

A stepwise iterative process is proposed in this paper, which includes five steps:

(1) concerns elicitation

Through reading and analyzing requirement statements carefully, behaviors, objects, goals, and properties can be elicited, some of which will be identified as concerns. This step can follow the idea of Theme/Doc method by utilizing nature language processor to identify different concerns from large scale and complex textual requirement statements [3]. Domain knowledge will be helpful in assisting concerns capturing.

(2) concerns & rough relation graph specification

With a set of concerns identified, these concerns will be specified. The formal definition of a concern includes a name, a piece of description, a set of related requirements, and a set of concepts it includes (concepts will be determined in step (3)). The dependency relationship among concerns will be recognized and specified as rough relation graph.

(3) concepts identification

For each concern and those requirements that it relates to, several concepts can be identified, which represent the key actions, objects or properties of that concern. Each concept can also be specified and added to the specification of the concern.

(4) variables definition and classification

For each concept and those requirements it resides, several variables will be defined and classified as monitored, controlled, input, output, term or mode. The classification of variables follows the ideas of SCR method.

(5) SCR requirement specification

In SCR approach, the system requirements are specified as a set of relations that the system must maintain. Three specifications are included in SCR requirement documents, they are system requirement specification(SRS), system design specification(SDS) and software requirement specification(SoRS). Detailed description of SCR requirement method can be found in [1].

a) SRS specification

SRS describes the external behavior of the system in terms of monitored and controlled quantities in the system environment. For all the monitored and controlled variables, we can trace back to those requirements where they were identified, and from these requirements, the relationship among monitored and controlled variables will be defined as tables.

* This work is supported by China "863" High-tech Research and Development Project under Grant No. 2007AA01Z123.
** This paper is an extended abstract of [2].

b) SDS specification

SDS identifies and document input and output devices. For all input and output variables, we can trace back to those requirements where they were identified, and the relationship between input and output variables and the monitored and controlled variables.

c) SoRS specification

SoRS refines SRS by adding estimation of how to use input variables to calculate monitored values and how to derive output values from controlled values. Therefore, the correspondence between input variables and monitored variable, output variables and controlled variables will be specified.

d) Dealing with hardware malfunction and timing constraints

Finally, behaviors related to hardware malfunctions and timing constraints will be added.

The process defined above will be done iteratively until validation of intermediaries has been conformed. Concerns and their specification as well as concern relationship will be checked, negotiated and finally confirmed by different stakeholders together, and such checks as coverage, consistency and completeness of SCR specification can be conducted by tracing requirement statements, concepts, key variables and constructs in SCR specification.

To demonstrate and evaluate our method, we have applied it to a classical case – the Light Control System by specifying Hallway Section requirements[4]. It is indicated that our approach provides a guideline for bridging the gap between textual requirement definitions and formal SCR requirement documents via concerns, and facilitates applying aspect oriented approach to finding and solving conflicts in requirements.

Case studies have been given and discussed on SCR method[5], however, concerns have not been considered during deriving SCR specification from textual requirement definition.

Many works have been done on separation of concerns in requirement analysis and modeling [3,6,7]. All of these works focus on identifying and specifying concerns from either textual requirement definition or UML specification, especially AORE is for identifying and composing crosscutting concerns. Concerns are used to modularize requirements, not for generating formal requirement documents.

Concept-based approach for requirement analysis and modeling is proposed in [8], which utilizes formal concept analysis for finding or delivering class candidates from a given use case description.

Our approach is inspired by [3] for concern identification in prose requirement statements and [8] for using concepts to model requirements, however, our approach focuses on producing formal requirement specification from textual requirement definition with concerns and concepts as the intermediaries.

Lack of tool support will make it hard to apply our approach to a big practical project. Therefore, a prototype for specification of concerns and their relationship is under development in our lab. In the future, we would like to use formal concept analysis to help validation of concerns specification and requirement specification, conduct more case studies to real systems by utilizing supporting tool so that some quantitative results will be generated on the benefits of our approach.

References

[1] C. Heitmeyer. Formal Methods for Specifying, Validating, and Verifying Requirements. J. UCS 13(5): 607-618 (2007)

[2] Ying Jin, Jing Zhang, Weiping Hao, Pengfei Ma. Concern Based Approach to Generating SCR Requirement Specification: a Case Study, accepted by CSIE2009, March 31 - April 2, 2009, Los Angeles/Anaheim, USA.

[3] Baniassad, Clarke, Theme: An Approach for Aspect-oriented Analysis and Design[C]//Proceedings of the 26th Int'l Conf. on Software Engineering, Edinburgh, Scotland. 2004: 158-167.

[4] The light control case study: Problem description. Journal of Universal Computer Science, Special Issue on Requirements Engineering ,Vol. 6, 2000.

[5] C. Heitmeyer, R. Bharadwaj. Applying the SCR Requirements Method to the Light Control Case Study. Journal of Universal Computer Science (JUCS), August 2000.

[6] S. M Sutton Jr. Early-Stage Concern Modeling. Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, April, 2002.

[7] E. L. A. Baniassad, P. Clements, J. Araujo, A. Moreira, A. Rashid, and B. Tekinerdogan (2006) Discovering Early Aspects. IEEE Software. Volume 23, Pages 61-69.

[8] W. Hesse and T. Tilley. Formal Concept Analysis Used for Software Analysis and Modeling. In B. Ganter, G. Stumme and R. Wille, editors, Formal Concept Analysis: Foundations and Applications, pages 288-303. Springer-Verlag, Germany, 2005.

User-oriented Preparative Treatments for Requirements Engineering

Fei He[1]
Yoshiaki Fukazawa[2]

Department of Computer Science and Engineering
Waseda University,
Okubo 3-4-1, Shinjuku-ku, Tokyo, Japan

Abstract
In order to resolve the classical problems (the major difficulties are the user needs and for developers to understand those needs) of requirements engineering, we propose a 3-phase user-oriented preparative treatment to generate clear user requirements and offer those user requirements as high quality input resources for the later generation of system requirements: phase 1, to build up users' user information reserves. Only when we put an application system in the background of users' whole systematic and informationalized reserve assets, rather than treat it as an independent or solitary entity, can we fully understand the assigned roles and positions of the application system, and only in this way can we correctly anticipate the way of evolutions of the application system and realize the essence of its requirements; phase 2, to generate user requirements which are descriptions of states and problems of certain business application, and are independent to system requirements which focus on solutions for system implementation; phase 3, to generate system requirements by using user requirements, and build up the linkage between user requirements and system requirements.

Keywords: user-oriented preparative treatment, user information reserve, user requirements.

1 Introduction

There are already many approaches and efforts being applied in requirements engineering, and Betty Cheng and Joanne Atlee provides a profound and comprehensive summary work for requirements engineering in [1]:

- In the phase of requirement elicitation, there have methods like goals [2-8], policies [9], scenarios [10], anti-models [11][12], nonfunctional requirements [13], behavioral models [14][15], domain-specific descriptions [16];

[1] Email: hefei@ruri.waseda.jp
[2] Email: fukazawa@waseda.jp

- In the phase of requirement management, there are methods like scenario based [17], feature based [18], traceability based [19][20][21] managements.

(As to phases like requirement modeling, analysis, validation & verification, they are not the focus of our discussions here, we just put them aside).

As the history of Standish Group CHAOS Reports (1994-2006) reveals: software project success rates have been improved greatly, but the rate of required features and functions making it to the released product has no much improvement at all, we can say we are still facing the old problems our predecessors engaged: the major difficulties in software developments are the user needs and for the developers to understand those needs.

The above mentioned methods and techniques are certainly great achievements for requirements engineering. But why we do not have much significant progress in all these years.

To clarify the problem, first let us have a look on the ideal definitions of requirement and RE themselves: requirements represent guidelines which allow developers to deliver agreed upon software system, and RE (especially requirements acquisition) must hold user-centric operations which are based on the user's perspectives, and the processes shall be the combination of the human performance engineering and business process reengineering together with the support of information engineering [22]". But the reality is: most people leave the latter part (user-centric) alone and keep busy on the former part (guidelines to software system), even some researches titled with "user-centered" are only proposing methods to squeeze system requirements from users, and then all those efforts result in requirements which are not of users, but of developers. Thinking exactly the reverse, what good it is if it does not fulfill the intended roles or usages.

Then let us check the other aspect of requirements: the lifetime of requirements. Requirements are evolutionary by nature; there is need for continuous requirements management. It is important to maintain complete and easily understandable (by all stakeholders) requirements documentation at all times. In real cases, most developers and even users only care about immediate interests: "On time, on budget, and on scope", with little regards on long-term managements.

Here we come to the conclusion: people are doing right for the right things (the right doing are the above listed methods and techniques, the right things are the requirement specification for software systems); and this is also the problem: we all start from the standpoints of software engineering, from the software systems and for the software systems, but we still do not take the following problems serious: what is between the needs in users' minds and the corresponding requirement specification, where those requirements really come from after all (they are already there in users' daily lives or just because of the needs for developing a system). With the problems left unsolved, we will always face the poor input situations of requirements, and will never have any real advancement.

2 The Preparative Treatments

First, we list the classical problems (*P*) in RE and our recommendations (*R*) for them respectively:

I. As to problems on the quality of requirements

P-a: The current roles of software developers have been intertwined with those of users. Instead of providing systematic solutions, software developers usually take too much effort, in some cases even have to live with users, to acquire users' requirements, for most users have no time, patience or ability to clearly express their requirements or for the developers do not think users have this

kind of ability. This usually lead to the fact that the requirements captured by software developers from users do not represent users' original intentions but only software developers' subjective interpretation of users' requirements. Also people do not tell users' requirements apart from requirement specifications which are in fact system requirements.

R-a: A clear line should be drawn between the work roles of users and software developers and between the artifacts of users and developers by adding certain user-oriented preparative treatments into the traditional requirements engineering. Within the period of the preparative treatments, users make their own artifacts (user requirements) which are descriptions of states and problems of target business applications and are independent to system implementations; then developers could continue to develop their own requirement specifications (system requirements) for executable technical solutions from those user requirements.

P-b: Users are usually unenthusiastic or uncooperative on requirements, for they think it is useless to their business especially when requirements engineering is over and it should be developers' duties to specify users' requirements, and for the fact that current requirements are only for system building which is out of users' business.

R-b: Users should be conscious and aware that a clear representation of their requirements will not only help to achieve the target application system but also help them understand their own business better.

P-c: Even if users are cooperative about requirements engineering, they do not have technical support to help them represent their requirements while developers are surrounded by all kinds of languages, development environments and methodologies.

R-c: Users should be provided with various technical supports such as integrated tools, environments, methodologies and training throughout the period of user-oriented preparative treatments. This aims to have user requirements formally organized, represented and documented, which will naturally make the transformation between user requirements and system requirements much easier and more efficient.

II. As to problems on the changes of requirements

P-d: Due to the limited resources and increase in complexity, most developers and users will not spare much effort to consider the changing nature of requirements in a project. Most projects care most for the immediate interest instead of overall view or a long-term plan, which make the projects lack of the flexibility to be responsive to the changing environments.

R-d: Instead of focusing on the immediate interest, we should enlarge our vision to all information about users (including organization, business processes or patterns, resources, etc) to establish a user information reserve for any future requirements and for preparation to any changes of requirements. The process of establishing the user information reserve should be integrated into users' daily work in a long-term period.

Aiming at the above problems and recommendations, we propose a 3-phase user-oriented preparative treatment for users, not software developers, to accurately generate users' own requirements. With appropriate training and supports of information techniques, those user requirements will be more effective and thus better transformed to system requirements.

Phase 1: Generation of User Information Reserve

The key problem in capturing requirements is the uncertainty of requirements. The uncertainty of requirements is the direct result of uncertainty of users' knowledge. The primary factors of uncertainty of knowledge are things like: the carriers of knowledge (normally languages) are uncertain and the general knowledge (or common sense) is different in different fields. So we need to provide proper languages or tools to capture and represent knowledge that around users' daily and routine works and make them informationalized. Those users' information may come from every aspects which may include organization (structure, etc), business processes (routine transaction processing, budget allocation, etc), internal resources (human labors, hardware, storages, time, etc.), external resources (customers, partners, law, social obligations, etc). Users should try their best to organize and represent all user information in a systematic way for any future requirements, and the result is the user information reserves.

For example, a user information reserve for logistics businesses may include a UML (Unified Modeling Language)-based diagram of organizational structure, a BPML (Business Process Modeling Language)-based representation of logistics workflow, a structured form-based list of specifications of computer terminal possessed by a service point, and an overall graph describing how to allocate resources and assign work roles according to the workflow of a logistics business.

Only when the assigned information system is placed in the holistic user information background, can it be possible to fully understand the roles and positions of the assigned application system, anticipate the way of system evolution, and realize the essence of the later user requirements.

Summary	
Input:	all the information around users;
Output:	user information reserve (the systematic representation of user information);
Techniques:	human performance engineering and business process reengineering together with the supports from information engineering [22].

Phase 2: Generation of User Requirements

In this phase, users will generate their own requirements which are descriptions of states and problems of certain business applications, and are independent to system requirements which focus on solutions for system implementation.

To generate user requirements, users have to first define the context of the target application system. A context is a specialized environment which outlines the capability of the target application system. For example, a context may include specified customers, given goals and confined resources. Then users have to extract the relevant info from user information reserves and organize this info according to the defined context to form the user requirements. The exact operations in the generation process may include the following steps:

- Negotiating the acceptance criteria (priority, etc.) among stakeholders;
- Defining the detailed system boundary (participants, goals, functions, resources, etc, and their relationships);

For example, a Japanese logistics service company plans to expand auto parts distribution business to Chinese markets. In addition to the existing user information reserves (organization of headquarters in Japan, current markets, typical logistics service flows or other knowledge and experience developed over the years), the company needs to collect and organize the info on all elements in the new business (from payment systems of local labor, tax, tariff duties of target areas, funds, local partners and

customers, local transportation networks and throughput of storages, local laws and regulations, local standards of auto motor industry, environmental specifications, weather, social images, brand awareness, etc) into a new user information reserves. From all the user information reserves (the existing and the new), you may select the relevant info (such as company's typical logistics service flows, local partners and customers, local transportation networks and throughput of storages, and local laws and regulations) as the context of the new business application system for Chinese market. Although the above process is a time-consuming and laborious work, it makes users have a clear understanding over the new market, provide supports for decision-making and ensures the success of the expansion plan to the details.

The changes of the context lead to the changes of requirements and the changes of the business application system. By analyzing and anticipating the possible changes of context in advance, such as performing scenario simulation/evaluation and preparing counter-plans, users can improve the flexibility of their requirements.

Example on the influence of changes of context could be that the new tendency of product Eco-responsibility ordinances on auto motor industry (policy rewards on small car with cylinder volume < 1.5liter and restrictions on large non-fuel-efficient vehicle with displacement > 500g/km) may lead to the changes of product models, then the orders of what you may ship and how you can ship (from packaging, routes of shipment, to means of shipping). If users have preparation in advance by analyzing the features and changes of relevant elements in context, they will surely not end in any abrupt results.

By performing and integrating phase 1 and phase 2 in users' daily wok, users will streamline organization structures and business processes, clarify their own strength and weakness, make short-term and long-term plan more efficient, and be more competitive in crisis or changing environment.

Realizing the above mentioned benefits, users may be interested in the requirements related operations and would be willing to and be able to reuse their requirements and take most efforts to continuously renew the user information reserves.

Summary

Input:	user information reserves from phase 1 and specific context for target business application system;
Output:	user requirements;
Techniques:	in addition to the existing methodologies and techniques in requirement elicitation, modeling and analysis (such as scenario based [17], feature based [18] methods), we add context managements to reflect the capability and transition of application system.

Note: The target business application system mentioned in phase 2 is still staying on the level of pure business, rather than a software system.

Phase 3: Generation of System Requirements

This phase aims to derive system requirements from user requirements under the conditions (such as available information techniques and resources for the project) proposed by developers, and to establish cause-effect relationships between user requirements and system requirements. Once changes happened on one side, the cause-effect relationships make it possible to trace the corresponding effects on any other side. Detailed works include:

- Find the access/intermediate points of binding from each side (users and developers);
- Make an agreement to clarify responsibilities on collaboration for each aspect of requirements from both parties (users and developers).

For example, one of users' requirements is tag based tracking service for customs clearance, position locating, and history recording for delivery efficiency evaluation. The

duty of user requirements shall be: providing info on the network of distributed service points (geo-location, number, distance, etc.). In the case of the number of service points changed, the first thing to check is whether the change is beyond the scope of agreement or not. If the change is beyond the scope of agreement, new agreement shall be reached on the agenda of next meeting.

Summary	
Input:	user requirements from phase 2 and conditions of system developers;
Output:	system requirements and linkages between user requirements and system requirements;
Techniques:	in addition to the existing methodologies and techniques in requirement elicitation, analysis and management (such as traceability based managements [20]), we propose binding for the smooth transformation from user requirements to system requirements.

The feasibility and flexibility of the user information reserves and the user requirements of the business application system can be evaluated by third-party consultants, then according to the evaluation, users can decide when to implement the system and who shall have the implementation contracts. Also with a clear linkage to system requirements, users can even change developers when the project falls in trouble. All this makes customers much more independent to system vendors compared to the current prevailing fact that once a customer signed a contract with a certain system vendor, the customer will have no choice but rely on the same system vendor for the lifetime.

The things after the phase 3 are the traditional works of requirements engineering: to generate requirement specification for later software developments.

3 Conclusion

In order to resolve the classical problems (the major difficulties are the user needs and for developers to understand those needs) of requirements engineering, we propose a 3-phase user-oriented preparative treatment to generate clear user requirements and offer those user requirements as high quality input resources for the later generation of system requirements: phase 1, to build up users' user information reserves. Only when we put an application system in the background of users' whole systematic and informationalized reserve assets, rather than treat it as an independent or solitary entity, can we fully understand the assigned roles and positions of the application system, and only in this way can we correctly anticipate the way of evolutions of the application system and realize the essence of its requirements; phase 2, to generate user requirements which are descriptions of states and problems of certain business application, and are independent to system requirements which focus on solutions for system implementation; phase 3, to generate system requirements by using user requirements, and build up the linkage between user requirements and system requirements.

According to our capability, we follow the roadmap of first phase 2, then phase 3, finally phase 1 to unfold our research. The 3 phases cover a quite huge range, especially the phase 1 which concerns with techniques beyond the information techniques in our hands. This needs more efforts on cross-cutting works and cooperation from various communities.

References

[1] Betty H. Cheng, Joanne M. Atlee. Research Directions in Requirements Engineering. In Proceedings of International Conference on Software Engineering, 2007, Future of Software Engineering, pp. 285-303.

[2] A. van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In Proceedings of the IEEE International Requirements Engineering Conference, 2001, pp.249–263.

[3] C. Alves and A. Finkelstein. Challenges in COTS decision-making: a goal-driven requirements engineering perspective. In Proceedings of the International Conference on Software Engineering and Knowledge Engineering, 2002, pp.789–794.

[4] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In Proceedings of ACM SIGSOFT Foundation on Software Engineering, 2002, pp.119–128.

[5] B. Gonzalez-Baixauli, J. C. S. do Prado Leite, and J. Mylopoulos. Visual variability analysis for goal models. In Proceedings of the IEEE International Requirements Engineering Conference, 2004 , pp.198–207.

[6] E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In Proceedings of ACM SIGSOFT Foundation on Software Engineering, 2004, pp.53–62.

[7] R. Settimi, E. Berezhanskaya, O. BenKhadra, S. Christina, and J. Cleland-Huang. Goal-centric traceability for managing non-functional requirements. In Proceedings of the IEEE International Conference on Software Engineering, 2005, pp.362–371.

[8] S. Liaskos, Alexei, Y. Yu, E. Yu, and J. Mylopoulos. On goal-based variability acquisition and analysis. In Proceedings of the IEEE International Requirements Engineering Conference, 2006, pp.76–85.

[9] T. D. Breaux and A. I. Anton. Analyzing goal semantics for rights, permissions, and obligations. In Proceedings of the IEEE International Requirements Engineering Conference, 2005, pp.177–188.

[10] N. Maiden and S. Robertson. Developing use cases and scenarios in the requirements process. In Proceedings of the IEEE International Conference on Software Engineering, 2005, pp.561–570.

[11] G. Sindre and A. Opdahl. Templates for misuse case description. In Proceedings of the International Workshop on Requirements Engineering, Foundations for Software Quality, 2001, pp.125–136.

[12] A. van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In Proceedings of the IEEE International Conference on Software Engineering, 2004, pp.148–157.

[13] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. Nonfunctional Requirements in Software Engineering. Kluwer, 1999.

[14] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. IEEE Trans. on Software Engineering, 29(2):99–115, 2003.

[15] S. Uchitel, J. Kramer, and J. Magee. Behaviour model elaboration using partial labelled transition systems. In Proceedings of ACM SIGSOFT, Foundations on Software Engineering, 2003, pp.19–27.

[16] H. Kaiya and M. Saeki. Using domain ontology as domain knowledge for requirements elicitation. In Proceedings of the IEEE International Requirements Engineering Conference, 2006, pp.186–195.

[17] T. A. Alspaugh and A. I. Anton. Scenario networks for software specification and scenario

management. Technical Report TR-2001-12, North Carolina State University at Raleigh, 2001.

[18] M. O. Reiser and M. Weber. Managing highly complex product families with multi-level feature trees. In Proceedings of the IEEE International Requirements Engineering Conference, 2006, pp.146–155.

[19] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In Proceedings of the IEEE International Requirements Engineering, 2005, pp.135–144.

[20] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. IEEE Trans. on Software Engineering, 32(1):4–19, 2006.

[21] R. Settimi, E. Berezhanskaya, O. BenKhadra, S. Christina, and J. Cleland-Huang. Goal-centric traceability for managing non-functional requirements. In Proceedings of the IEEE International Conference on Software Engineering, 2005, pp.362–371.

[22] Karen L. McGraw, Karan Harbison. User-centered Requirements: The Scenario-based Engineering Process. Lawrence Erlbaum, Mahwah, N.J., 1997.

The Specification Construction of a Service-Oriented System Using the SOFL Method

Weikai Miao[1]

Department of Computer Science
Hosei University
Tokyo, Japan

Shaoying Liu[2]

Department of Computer Science
Hosei University
Tokyo, Japan

Abstract

Service-oriented computing shows a great potential of achieving high productivity and low cost, but it faces a challenge in efficiently and correctly using existing services in producing a new service and ensuring its reliability. Building a formal model using a formal specification language allows the developer to thoroughly understand what existing services are needed for the new service, but how to construct the model so that it can effectively facilitate the developer to recognize the situations where existing services can be fully utilized still remains an open problem. In this paper, we describe an approach to applying the SOFL formal engineering method to the modeling of a service-oriented computing system by means of a case study. In particular, we focus on the issue of how to apply the SOFL three-step modeling approach to the construction of a formal specification for a web-based software service, exploring the general principle and specific techniques for using existing services in developing a new service model. The potential benefits and further challenges of our approach are discussed.

Keywords: Service-oriented Computing, Formal Engineering Method, SOFL

1 Introduction

The service-oriented computing (SOC) paradigm is promoting both the theory and technology of software development by using web services to support the development of low-cost, interoperable, evolvable, and massively distributed applications. Although the industrial world has developed a number of standards to formalize the specification of web services, developing web service-based systems is still a challenging problem for the lack of a comprehensive engineering method [11][15].

[1] Email: weikai.miao.x1@gs-cis.hosei.ac.jp
[2] Email: sliu@hosei.ac.jp

The dilemma for SOC software development is that informal engineering methods are friendly to developers but have a tremendous drawback in quality assurance and maintenance due to ambiguities in requirements specifications and system models. On the other hand, formal methods, such as Z [3], VDM [4], and B method [2] offer great potentials of ensuring the correctness of software systems through formal specification, refinement, and verification. It is obvious that formal methods can make a considerable software quality assurance if they are successfully applied in practice [1][12]. Unfortunately, very few practitioners could use them in practice [9] because of their complexity and technical difficulties for large scale system development. It is a challenge for practitioners to construct such kind of formal model in a development process to facilitate them in easily understanding what existing services are available and needed for the new application, and how they can be coherently integrated into the new application under construction.

To address this problem, we try to adopt the SOFL (Structured Object-Oriented Formal Language) formal engineering method into the SOC environment. The researches on the SOFL method suggest that it is a practical software engineering method that maintaining good balance between formal methods and human activities in software development processes, which allow practitioners to easily apply [6][10][13][16]. In this paper we describe a novel application of the SOFL three-step modeling approach to developing a formal specification for an air ticket reservation system in service-oriented manner. Using the three-step approach of SOFL, a formal design model can be effectively achieved through building informal, semi-formal, and then formal specifications. We aim at exploring and deriving useful guidelines for integrating existing services into the process of developing a formal specification using the SOFL formal engineering method through the case study.

The rest of the paper is organized as follows. Section 2 briefly introduces the SOFL method, particularly the idea of the three-step modeling approach. Section 3 describes the domain background of the case study. Section 4 presents the detailed process of specification construction using the SOFL three-step modeling approach. Section 5 discusses our experience gained from the case study. Finally, in Section 6 we conclude this paper and point out future research directions.

2 Brief Introduction to SOFL

The SOFL formal engineering method offers a systematic way to integrate existing formal methods into conventional software engineering techniques. Basically, it advocates three integrations for different purposes. The first is to integrate the VDM specification language (VDM-SL) with classical data flow diagrams to form an intuitive, rigorous structuring method for building comprehensible formal specifications. The second is to integrate formal refinement principle with evolution principle to establish a practical transformation model for developing a formal specification into a satisfactory implementation. The third is to integrate formal proof theory with software inspection and testing principle to build rigorous inspection and testing techniques [13][7][5].

The three-step modeling approach advocates the building of a formal model for a software system through the constructions of informal, semi-formal, and then

formal specifications. This approach provides precise guidelines for creating the informal specification for requirements acquisition, transforming it into a semi-formal specification, and then developing it into a formal specification for design. The informal specification is aimed at capturing the abstract but complete aspects of the desired functions the system is expected to supply, the data resources needed to fulfill the functions, and necessary constraints on both the functions. The semi-formal specification is expected to refine the informal specification into a more precise and structured document so that communication between the user (or client) and the developer can be effectively realized. The feature of the semi-formality is reflected by the precise declarations of data types and variables, and the informal descriptions of invariants and process specifications in each module. The formal specification is intended to precisely define the architecture of the whole system and each of its components. The architecture is represented by a formalized data flow diagram, called *condition data flow diagram* (CDFD), and each process used in the CDFD as its functional component is specified using formally expressed pre- and post-conditions in a module. Apart from these, all type and state invariants are also formalized using the SOFL notation.

3 Domain Background of the Case Study

We take an *online air ticket reservation system* (OATRS) as the domain to apply the SOFL three-step modeling approach. An OATRS is intended to provide services for customers to register and manage their personal information, search, reserve, cancel, and change air tickets through a travel agent. A complicated OATRS of a travel agent may also provide more other functions, but we deliberately limit our study in this paper to a simple OATRS because it is sufficient for us to explore the possibilities of the modeling approach and to derive necessary guidelines for utilizing existing services during the development process.

4 The Process of Specification Construction

4.1 Informal Specification

Following the advice of the SOFL approach, we build the informal specification of OATRS in three sections: functions, data resources and constraints. The desired functions are described briefly as operations in English and some complex operations are decomposed into more detailed sub-operations for understanding. The section of data resources provides a listing of all data items that are necessary for the realization of the operations and may be shared by those operations. A collection of constraints reflecting policy restrictions on operations or data resources is provided.

The above components of an informal specification are usually derived on the basis of a thorough analysis by the developer in collaboration with the client. The client is supposed to be familiar with the domain knowledge, while the developer is assumed to be equipped with existing services and is usually good at finding desired requirements through communication with the client. In order to integrate some web services that can be utilized in the target system, the developer and the client first discuss the requirements. As a result, the developer gains a thorough

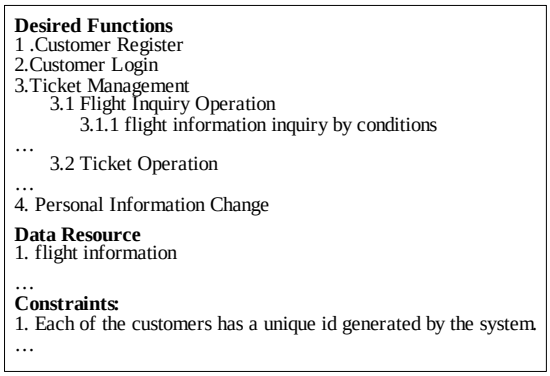


Fig. 1. Informal specification of the Online Air Ticket Reservation System.

understanding of the requirements and writes them down in a modular fashion. Then, the developer proceeds to search related software services. The purpose of the searching in this stage is to bring relevant candidate services for further examination on the appropriateness of being deployed in the system. The developer may also realize the necessity to modify the informal specification by, for example, reorganizing the functions so that the related services can be mostly reused. This is usually an evolutionary process.

On the basis of the initial informal specification, we gained the understanding that operations 3.1.1 and 3.2 can be implemented by existing web services. We then created necessary operations or modified existing operations. Figure 1 shows the major parts of the informal specification for our OATRS which results from the above actions.

By now an abstract understanding of the system functions and the reusable services have been achieved. Since the informal specification are ambiguous, our understanding about the system and reusable services may still be limited. Since the informal specification are ambiguous, our understanding about the system and reusable services may still be limited. We need to refine the specification into a semi-formal specification.

4.2 Semi Formal Specification

Communication between the developer and the client in capturing requirements plays an important role in software development, especially for developing service-oriented systems. The developer also needs to help the client understand the information of the selected web services. Semi-formal SOFL specification facilitates good communication between clients and developers. After finishing the informal specification, the developer may not be able to decide which function can be implemented by which service clearly because the functions in the specification are not defined precisely enough. Our solution is to refine the specification into semi-formal one by construction together with the further service selection. During this process, the specification is refined into a semi-formal one and the sets of candidate services for the functions may also be reduced. This process is an intellectual work that requires the integration of developer's knowledge and experience, the understanding of the services, and the communication with the client.

```
<complexType name="FlightInformation">
...
<wsdl:message name="FlightSearchRequest">
…
<wsdl:operation name="FlightSearch" parameterOrder="…">
<wsdl:input message="impl:FlightSearchRequest" name="FlightSearchRequest" />
<wsdl:output message="impl:FlightSearchResponse" name="FlightSearchResponse" />
</wsdl:operation>
```

Fig. 2. A segment of the WSDL file for "Flight Search Tool" Service.

```
FlightSearchRequest = composed of
                        flight_code: string
...
FlightSearchResponse: set of FlightInformation;
FlightInformation=composed of
                        flight_code: string
...
process FlightSearchTool(search_con: FlightSearchRequest)flight_result: FlightSearchResponse
post if the searching condition is input , the flight information fits the conditions should be listed
end_process
```

Fig. 3. SOFL Specification of the "Flight Search Tool".

In addition to the original SOFL principle for building semi-formal specifications[13], the developer need to analyze the WSDL (Web Service Description Language) or BPEL (Business Process Execution Language) files of the services to understand the precise inputs and outputs information of the services (including the types and numbers of the parameters). Together with this action, the functions which may be implemented by services are defined as SOFL processes so as to utilize the most appropriate services. The developer and the client discuss and decide the association between the SOFL process and the service during this process. However, in this stage the selection of services is mainly based on the data characteristics of the inputs and outputs information. To precisely verify the functions of the services, the developer may use testing on the basis of a formal specification to be built later on.

Let us take the "*flight information inquiry*" function component (function 3.1.1 in Figure 1) as an example to illustrate the service selection. After analyzing the WSDL files of the candidate services and understanding the requirements, we attempt to use the service "*FlightSearchToolService*" to implement our desired function. A segment of the WSDL file of the service is shown in Figure 2.

From the WSDL file of this service, we can extract the data structures of the input and the output messages. And then we can further clarify our function on the basis of these pieces of information. The function is abstracted into a SOFL process and its related data structures of inputs and outputs are defined. Figure 3 shows the semi-formal specification of the process.

Since the semi-formal process to be implemented using a service results from the collaboration of the developer and the client, the consistency between the SOFL process and the service needs to be checked. This consistency focuses on the data structures, including the types and the number of variables. To illustrate the checking procedure, we need to define necessary concepts, terms, and notation.

Definition 4.1 Let S denote a web service. S can be abstracted as: $S = (InputMsgSet, OutputMsgSet, ConsSet)$.

(i) $InputMsgSet(S) = \{inMsg_1, ..., inMsg_n\}$ defines the set of the input mes-

sages belonging to the operations provided by S where $inMsg_i$ $(i = 1, ..., n)$ is an input message. Each *input message* is defined as: $inMsg = \{v_1, ..., v_m\}$ where v_j $(j = 1, ..., m)$ is a variable.

(ii) $OutputMsgSet(S) = \{outMsg_1, ..., outMsg_h\}$ defines the set of the output messages belonging to the operations provided by S where $outMsg_i$ $(i = 1, ..., h)$ is an output message. Each *output message* is defined as: $outMsg = \{v_1, ..., v_k\}$ where v_j $(j = 1, ..., k)$ is a variable.

(iii) $ConsSet = \{cons(v_1), ..., cons(v_p)\}$ defines the set of the constraints on each variable v_j $(j = 1, ..., p)$ in each message where $cons(v_j)$ returns a predicate expression of the constraints on v_j.

Definition 4.2 Let P denote a SOFL process. Then P can be abstracted as $P = (InPortSet, OutPortSet, InvSet)$.

(i) $InPortSet(P) = \{inPort_1, ..., inPort_f\}$ defines the set of input ports of P where $inPort_i$ $(i = 1, ..., f)$ is an input port. Each *input port* is defined as: $inPort = \{v_1, ..., v_g\}$ where v_j $(j = 1, ..., g)$ is a variable in this port.

(ii) $OutPortSet(P) = \{outPort_1, ..., outPort_q\}$ defines the set of output ports of P where $outPort_i$ $(i = 1, ..., q)$ is an output port. Each *output port* is defined as: $outPort = \{v_1, ..., v_r\}$ where v_j $(j = 1, ..., r)$ is a variable in this port.

(iii) $InvSet = \{inv(v_1), ..., inv(v_t)\}$ defines the set of the invariants on each variable v_j $(j = 1, ..., t)$ in each port where $inv(v_j)$ returns a predicate expression of the invariants on v_j.

Definition 4.3 Let V be a set of variables and $v \in V$. Then, we define a function $Type$ over V such that $Type(v)$ returns the data type of v.

Definition 4.4 Let V_1 and V_2 be two sets of variables. Then, a relation $\simeq$ is defined as follow:

$\simeq: V_1 * V_2 \rightarrow boolean$

$x \simeq y \stackrel{\triangle}{=} Type(x) \subseteq Type(y)$ where $x \in V_1$ and $y \in V_2$.

This definition represents the fact that any variable x in V_1 has the relation $\simeq$ with a variable y in V_2 if and only if the data type of x is a subset of the type of y.

Based on the above related definitions, the final binding between a SOFL process and its corresponding selected service should satisfy the following criteria:

Criterion 4.5 *Let P denote a process and S as its selected service. Then, S and P must satisfy the following condition:*

1.1 $\forall_{inPort \in InPortSet(P)} \forall_{x \in inPort} \bullet \exists_{inMsg \in InputMsgSet(S)} \exists_{x' \in inMsg} \bullet x' \simeq x \wedge inv(x) \Leftrightarrow cons(x')$

1.2 $\forall_{outPort \in OutPortSet(P)} \forall_{x \in outPort} \bullet \exists_{outMsg \in OutputMsgSet(S)} \exists_{x' \in outMsg} \bullet x' \simeq x \wedge inv(x) \Leftrightarrow cons(x')$

This criterion ensures three things. First, for every input or output port of process P, there exists a corresponding input or output message belonging to service S; for every variable in the data flows linking to those ports of P there exists a corresponding variable in the those messages of S; and all the corresponding

```
<xsd:simpleType name="address">
<xsd:restriction base="xsd:string">
<xsd:minLength value="10" />
<xsd:maxLength value="150" />
```

Fig. 4. The Constraints in WSDL File.

```
inv
forall[x:Customer]|len(x.address)>=10 and len(x.address)=<150
```

Fig. 5. The Constraints in SOFL Specification.

variables satisfy the relation "$\simeq$". Because for some XML data types there may not be the exactly same data types in the SOFL formal language, we use the notation "$\simeq$" to express the equivalence of data types.

The data structure consistency between the process "*FlightSearchTool*" and service "*FlightSearchToolService*" can be checked according to Criterion 4.5. In our example, all the data types of the variables in the input data flow "*FlightSearchRequest*"and the output data flow "*FlightSearchResponse*" are consistent with the corresponding ones in the input message "*FlightSearchRequest*" or the output message "*FlightSearchResponse*" of the service. The consistency of the data type constraints are also checked. For example, the length of the variable "*address*" is defined between 10 and 150 characters in the WSDL file, which is showed in Figure 4.

At first we were not be aware of the constraints on the input variable "*address*" during the semi-formalization process. But the analysis of the services reminded us of this issue which provided us with an opportunity to explore further requirements. Then we can write the invariant which is shown in Figure 5.

Although we write the invariant formally in this example, the general principle of a semi-formal specification does not definitely require so. The characteristic of a semi-formal specification is that the data structures including data types and data stores are defined formally, but the invariants and *pre/post* conditions are kept informal.

Once the developer decides the web service which fits the inputs and outputs requirements of one process, he or she can draw the CDFDs showing to the client. The simple but clear CDFDs can help the client make judgements whether the system reflect their desired requirements with respect to the inputs and outputs. Sometimes a selected service is a composite service which is defined with both the WSDL file and the BPEL(Business Process Execution Language) file. Besides the data structures of the inputs and outputs, some control structures in the BPEL file are also transformed into corresponding ones in the CDFD. For example, another function "*Ticket Operation*" in our specification (function 3.2 in Figure 1) is bound with a composite service "*Ticket Operation Service*". After the analysis and information extraction, we semi-formalize the function as process "*TicketOperationService*" with a hierarchy CDFD reflecting the abstraction and internal detailed functions. The CDFD of its internal function is depicted in Figure 6 and the high level process is shown in Figure 7.

The CDFDs in this paper are drawn using the SOFL GUI editor tool our research group developed before [8]. Our experience suggests that drawing CDFDs, provide an initial image of the potential architecture of the system, and can therefore help the developer to gain a better understanding and to have a better communication

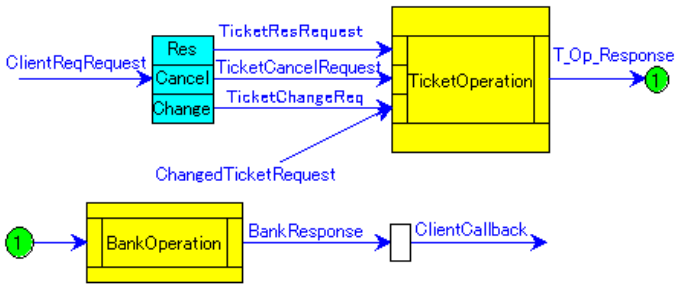


Fig. 6. CDFD of the Decomposition of "TicketOperationService".

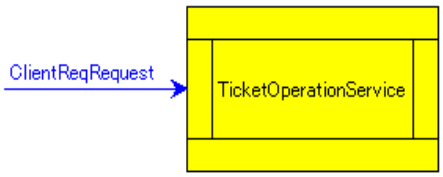


Fig. 7. CDFD of "TicketOperationService".

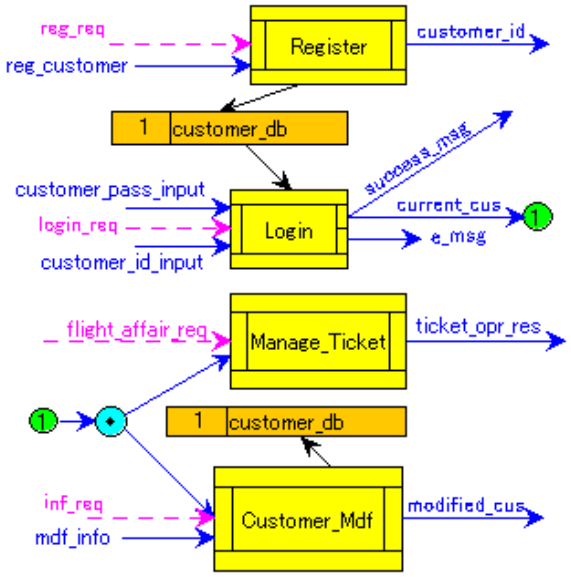


Fig. 8. Top-Level CDFD of the System.

with the client. As another example, we show the top-level CDFD in the semi-formal specification of the Online Air Ticket Reservation System in Figure 8.

Usually in this stage, developers cannot guarantee which service reflects the requirements accurately because they only judge it from the data types and informal descriptions. So in this stage, keeping the specification in semi-formal style is reasonable for further evolution or modification. We can test the web services by

```
process FlightSearchTool(search_con: FlightSearchRequest)flight_result: FlightSearchResponse
post bound(search_con)=>card(flight_result)=0
    or
    bound(search_con)=>card(flight_result)>0 and
    forall[x:flight_result]|x.flight_code=search_con.flight_code and
    ...
end_process
```

Fig. 9. The Formal Specification of Process "FlightSearchTool".

adding the *pre/post* conditions and then decide the final selected services. To carry out a precise design and to completely remove potential ambiguities in the semi-formal specification, the semi-formal specification needs to be transformed into a formal one.

4.3 Formal Specification

The main task of formal specification construction is to write the specification completely in the SOFL language. On the basis of the semi-formal specification, the developer can take the top-down or bottom-up approach to formalize each module. If we take the top-down approach, for example, all the informal parts, including invariants and *pre/post* conditions of the top-level module, should be formalized into the SOFL language. Moreover, for the development of service-oriented software, the developer should verify whether the selected services really correctly reflect the client's requirements. After adding the formal *pre/post* condition for each process, the developer can start the service testing. For example, Figure 9 shows the formalized process "*flight information inquiry*" with formal pre/post conditions.

We can take the approach in [14] proposed by our group to finish the testing.

The modules are organized into a proper hierarchy of CDFDs coupling with the corresponding hierarchy of modules. We choose the top level CDFD in Figure 8 as an example. This module contains several processes, including the *Manage_Ticket* process, which can be decomposed further. The main part of the top-level module in the formal specification is shown in Figure 10.

The second stage for the construction of formal specification is detailed design by transforming the processes in the specification into explicit ones. However, in our case the formal specification of each process used only simple predicate expressions, which were explicit enough for implementation. We therefore did not do the translation.

5 Advantages of the SOFL Method

Our experience in this case study has convinced us that writing a formal specification using the SOFL three-step modeling approach can significantly improve the communication quality and understanding of desired services by the developer. Meanwhile, we also found out some challenges to address in the future.

5.1 Strength

We found the strength of the SOFL modeling approach mainly in three aspects: simplicity, effectiveness, and expressive mechanism.

```
module SYSTEM_Ticket_Reservation
type
CustomerInf=composed of
                customer_name: string
                 …
                end
RegisteredCustomer=set of CustomerInf
…
var
ext  customer_db: RegisteredCustomer

inv
forall[x: CustomerInf] | not exists [y: CustomerInf] | x. customer_id =y. customer_id;
...
Behave CDFD_1;

process Login(login_req: sign, customer_id_input: nat0, customer_pass_input: string )
                current_cus: CustomerInf, success_msg: string | e_msg: string
ext rd customer_db
post (exists[current_cus: customer_db]| current_cus.customer_id= customer_id_input
     and current_cus.customer_pass = customer_pass_input)
     and success_msg="login succeeded"
     or
     (not exists[current_cus: customer_db]| current_cus.customer_id= customer_id_input
     and current_cus.customer_pass = customer_pass_input)
     and e_msg="incorrect password or id"
end_process;
     …
process Manage_Ticket (flight_affair_req: sign, current_customer: CustomerInf)
end_process;
…
end module;
```

Fig. 10. CDFD of the Top-Level Module.

5.1.1 ***Simplicity***

The SOFL language properly combines the formal notation VDM-SL with the intuitive graphical representation CDFDs. Such an integrated notation allows developers to write specifications with a good readability and programers to easily understand them. CDFDs present guidelines for both constructing formal specifications of the operations involved and understanding them by all the people involved in a project team. Due to the intuitiveness of the three-step modeling approach, we were given more chances to consider and analyze what existing services could provide helps in building the air ticket reservation system in our case study.

5.1.2 ***Effectiveness***

The three-step modeling approach addresses definite activities and criteria as milestones for the developer to follow during the process of specification evolution. This approach offers good communication and collaboration among different roles during the specification formalization process, which can effectively reduce the misunderstanding among them. Both the developer and the client can benefit from the semi-formalization of the specification, discovering some potential requirements and clarifying them. Moreover, the simplicity of the notations and clear structure of the

SOFL specification makes it easy to be implemented in those common programming languages. All these characteristics show the effectiveness of the SOFL method and language.

5.1.3 ***Expressive Mechanism***

SOC paradigm uses standard messages for the communication among web services. The encapsulation of a web service leads to good modularity of service-oriented systems. It is usually impossible to obtain the source code of existing services; only the information of data and interfaces in their WSDL or BPEL files are available. The SOFL language offers "*process*" which can be used to abstract the web services properly based on the limited description information without breaking the encapsulation of services. This is naturally suitable for service-oriented system modeling. No matter whether a service is atomic or composite, it can be abstracted into a single SOFL process or a hierarchical structure of SOFL CDFDs.

The unique CDFD itself is a well-defined diagram with rigorous syntax and semantics. It can intuitively express the corresponding SOFL specification, acting as a bridge between the professional developer and the client with little knowledge of software development.

5.2 Specific Guidelines

We have derived some specific guidelines for service-oriented modeling using the SOFL three–step modeling approach through the case study, which are described, respectively, below.

(i) **Distinguish the functions which can be realized by existing services from others.** In an informal specification, system functions are listed briefly, likely in a hierarchical structure. In this step, it is recommended to mark the functions which may be realized by potentially existing web services with comments. This will remind the developer to pay more attention to the functions when considering whether they should be decomposed into sub-functions. The decisions on decomposition have to be made on the basis of the developer's experience, knowledge of existing web services, and engineering judgement.

(ii) **Extracting web service descriptions correctly.** The extraction of web service description files is very critical in the process of specification construction. During the establishing of the semi-formal specification, the data types of the potential selected web services should be analyzed and transformed into valid types into the SOFL language.

(iii) **Creating CDFDs in semi-formal specifications.** Diagrammatic CDFDs can make the client understand the developer's design intention and judge whether the recommended web services by the developer satisfy his or her requirements. Our experience in the case study suggests that CDFDs can play an important role in helping communication between the developer and the client.

5.3 Challenges

(i) **Nonfunctional consideration.** The SOFL language is a formal language mainly used to express the functional requirements. The nonfunctional aspects such as security and response lantency of the web services plays an important role in service searching and selection. Some mechanisms need to be developed to express both functional and nonfunctional requirements in the SOFL specification.

(ii) **Accurate web service selection.** How to use the combination of the client's domain knowledge, the developer's experience and those existed techniques such as ontology base or knowledge base to explore the proper web services is not addressed completely. Accurate selection of web services in formalizing the SOFL specification for the service-oriented software is worth more research efforts.

6 Conclusion and Future Work

In this paper, we presented a case study applying the three-step modeling approach of the SOFL formal engineering method to the construction of a formal model for an air ticket reservation system in service-oriented manner. We have derived useful guidelines for systematically integrating web service descriptions into a new service system during the three-step approach. Our experience suggests that the SOFL formal engineering method be effective for service-oriented software development, but also shows some challenges in accurately determining reusable web services as software components. Our previous research has indicated a possibility of using specification-based testing to address this problem [14], but how to do it for web services in a distributed environment needs more research efforts.

As future research, we will make use of our experience gained from the case study to build a more effective service-oriented modeling approach, to study effective specification-based testing techniques for finding satisfactory web services, and to build efficient tool support for service-oriented application development.

References

[1] A. Hall. Using Formal Methods to Develop an ATC Information System. In M. G. Hinchey and J. P. Bowen, editors, *Industrial-Strength Formal Methods in Practice*, pages 206–229. Springer-Verlag, 1999.

[2] J-R.Abrial. *The B-Book*. Cambridge University Press, 1996.

[3] J.M.Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd, Second Edition, 1998.

[4] C.B Jones. *Systematic Software Development Using VDM*. Prentice Hall,Second Edition, 1990.

[5] Shaoying Liu. Integrating Specification-Based Review and Testing for Detecting Errors in Programs. In *Proceedings of The 9th International Conference on Formal Engineering Methods (ICFEM2007)*, pages 136–150. LNCS 4789, Springer-Verlag, Nov. 2007.

[6] Shaoying Liu, A.J.Offutt, C.Ho-Stuart, Y.Sun, and M.Ohba. SOFL: a formal engineering methodology for industrial applications. *Software Engineering*, (1):24–25, 1998.

[7] Shaoying Liu, Fumiko Nagoya, Yuting Chen, Masashi Goya, and John A. McDermid. An Automated Approach to Specification-Based Program Inspection. In *Proceedings of The 7th International Conference on Formal Engineering Methods (ICFEM2005)*, pages 421–434. K.K. Lau and R.Banach, Eds., Manchester, UK, LNCS 3785, Springer-Verlag, Nov. 2005.

[8] Shaoying Liu and Xiang Xue. Automated Software Specification and Design Using the SOFL Formal Engineering Method. In *Proceedings of 2009 World Congress on Software Engineering*. IEEE Computer Society, May 2009.

[9] D. L. Parnas. Education for Computing Professionals. *Computer*, 23(1):17–22, 1990.

[10] Hao peng Chen, Yao Shen, and Jian wei Jiang. Extended SOFL features for the modeling of middleware-based transaction management. In *Proceedings of 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 16–17. IEEE Computer Society, Jun. 2005.

[11] Michael P.Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40:38–45, 2007.

[12] S. Sahara. An Experience of Applying Formal Method on a Large Business Application (in Japanese). In *Proceedings of 2004 Symposium of Science and Technology on System Verification*, pages 93–100, Osaka, Japan, Feb. 4-6 2004. National Institute of Advanced Industrial Science and Technology (AIST).

[13] S.Liu. *Formal Engineering for Industrial Software Development Using the SOFL Method*. Springer-Verlag, 2004.

[14] S.Liu. Utilizing Formalization to Test Programs without Available Source Code. In *Proceedings of The Eighth International Conference on Quality Software(QSIC2008)*, pages 216–221. IEEE Computer Society, August 2008.

[15] W.-T.Tsai, Xiao Wei, Zhibin Cao, Raymond Paul, Yinong Chen, and Jingjing Xu. Process Specification and Modeling Language for Service-Oriented Software Development. In *Proceedings of 11th International Workshop on Future Trends of Distributed Computing Systems*, pages 181–188. IEEE Computer Society, May 2007.

[16] Jichuan Wang, Shaoying Liu, and Di Hou Yong Qi. Developing an Insulin Pump System Using the SOFL Method. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 334–341. IEEE Computer Society, Dec. 2007.

Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the ENTCS Macro Home Page.

Consistency of Networks of Components [1]

Zhiying Liu [2], David Lorge Parnas [3], and Baltasar Trancon y Widemann [4]

Software Quality Research Laboratory
Faculty of Informatics and Electronics
University of Limerick, Limerick, Ireland

Abstract

In this paper, we present an approach to describing a network of components and checking whether the network is complete and consistent, i.e. whether the components can work together properly. This model will be the basis of a procedure for checking the correctness of a network of fully specified components on the basis of their interface specifications and a description of the way that the components are connected. Our approach is different from others in that each component is viewed as a hardware-like device in which an output value can change instantaneously when input values change and all components operate synchronously rather than in sequence.

Keywords: Networks of components, completeness, consistency, delay-free loop.

1 Introduction

The most fundamental problem in software development is the complexity of the completed product; the way to deal with this complexity is a principle that is known as "Divide and Conquer", "Separation of Concerns" [5], "Abstraction" or "Information Hiding" [15]. A computer system should be decomposed into manageable modules that can be developed separately to reduce the complexity of the development task. Modularity separates the concern of dealing with the implementation of each module (while ignoring their interaction) from dealing with the interaction of the modules and their relationships without considering implementation details of any module.

However, separating concerns does not mean that the modules will be independent. It allows people to work on them separately but they will still depend on each other. When a software system comprises a set of components, they must communicate and work together harmoniously to perform the required task. In order to prevent developers from wasting time implementing components that will not work together, the consistency of the component interfaces and the correctness of this design with respect to its requirements should be checked early in the development.

[1] This work was supported by Science Foundation Ireland from 2002 to 2007 under an SFI Fellow appointment.

[2] Z. Liu's email: zhiying dot liu at ul dot ie.

[3] D. L. Parnas is Emeritus Professor at McMaster University, Canada and Adjunct Professor at UL. Email: David dot Parnas at ul dot ie.

[4] B. Trancon y Widemann is with Ökologie Modellbildung, Universität Bayreuth, Germany. Email: Baltasar at Trancon dot ie.

©2009 Published by Elsevier Science B. V.

1.1 Precise documentation of component interfaces

We consider documentation to be (at least) as important as the product itself [19]. Applying the information hiding principles requires that we document the interface information precisely without revealing any information that should be hidden. Only in this way can we be sure that the implementers of other modules do not use information about the internal structure and that the future changes may affect as few modules as possible. Consequently, preparing good module interface documentation is essential for improving the quality of our software.

TFM, the Trace Function Method, is the result of decades-long efforts [14], [16] to find a practical method for documenting module interfaces. It can be traced back to its precursor TAM [3], [22], [25]. It is a new approach to specify information-hiding module/component interfaces. For more details about TFM, readers are referred to [18]. Our work here is based on documents written using TFM. The essential property of this documentation is that the value of each output at a time t, is described as a function of the history of the input and output values up to and including time t.

1.2 Consistency checking of mathematical documents

We divide documentation checking into 1) checking of local properties in the documentation of one component, and 2) checking of global properties among related components. The first has relevance to this paper only in that the documents we are checking are assumed to have been subject to local checks previously.

Although mathematical documents are precise and unambiguous they are also highly detailed and oversights and other errors are quite common. To detect the early errors in such documents, we should validate them as early as possible [17]. The validation of design documents using pre-established criteria is a technical task that can only be carried out if the design document is precise enough to permit systematic analysis [19]. Research such as [10], [8], [9], [2], [27], [11] and [26] provide approaches to checking completeness and consistency of a individual document of one component.

In [6] DeRemer and Kron introduced and established the basic idea and concepts of module interconnection language (MIL), which was aimed to ensure consistency between different parts of a system. The idea was further developed in the work of [24] and [20]. But MIL-based consistency checking is limited to simple type checking, there is no semantics for defining module functions. In 1990s MIL was superseded by architecture description languages (ADLs) [7], [1], and the society tried to overcome the lack of semantics in the language and provide support for specifying both functional and non-functional characteristics of components. ADLs are intend to permit analysis of architectures and consistency checking of the structure descriptions. However there is no agreement on what constitutes an ADL, what aspects of an architecture should be modeled in an ADL, and which ADL is best suited for a particular problem [12], and certainly not a mathematical definition of the languages. Without solving these problems it is difficult for an ADL to be used as a basis for documentation checking.

Our general approach to checking conformance is the relational/functional

method from [19]. If the design documents are written in TFM, the individual tabular expressions in the specification of each component can be checked by table checking tools such as those introduced in [11] and [26]. In this paper, we assume that the interface specifications for each individual component have already been checked. Our concern is checking the consistency of the interconnected components and the conformance between a design and its specifications.

2 What are networks of components and why are they interesting?

The need to reduce complexity and increase flexibility has motivated the development of techniques for decomposing a system into modules and composing existing components back into a system. The assembly of a system made of components is made easier if the components are well defined and documented. If the system is not well organized and professionally documented, the task of building and maintaining the system is made more difficult than it would otherwise be.

2.1 Components, interfaces and the TFM interface specifications/descriptions

Components interconnect with each other through their interfaces. Before any further discussions we have to define what do we mean by "components" and "interfaces".

Definition 2.1 A software component is a collection of programs that is distributed as a unit, i.e. without modification, for use in larger systems [18]. A well-defined component should have clear interfaces for communication with its environment, and it must provide specified functionality. We treat a component as a hardware-like device in which an output value can change instantaneously when input values change.

Definition 2.2 The interfaces of a component are assumptions that the component makes about the others. An interface is generally an abstraction that a component provides of itself to the outside. It separates the methods of external communication from internal operation, and allows it to be internally modified without affecting the way outside components interact with it, as well as provide multiple abstractions of itself.

In related work we have used the TFM method [4], [23], [18] to document component interfaces. This approach differs from many earlier approaches that provide a relation with a domain of input histories and a range of output histories. In our approach the domain is a trace that includes both inputs and outputs in the past and the range is a set of possible values for the output. The work described in this paper is not only compatible with TFM but also compatible with any other method of providing component interface descriptions by documenting the relation between input/output histories and output values.

2.2 Interconnections and composition of components

If there are two communicating software components, A and B, B's interface to A is the weakest assumption about B that would allow you to prove the correctness of A. We need to check whether the assumptions of one component about the rest of the system have been met, whether the types of inputs match the types of outputs connected to them.

When components are assembled to perform a common task, there may be both explicit and implicit connections between them. The explicit connections are the invocations and procedure parameters of which the designers are conscious. There may also be explicitly shared variables used for communication. Sometimes however there are shared resources and variables that are part of the communication but overlooked by the designers and reviewers. It is essential to a useful analysis of the network of components that all connections are identified and included in the documentation.

Composing components means making a bigger component (we call it a compound component) by connecting a group of components together. These components are called the internal components of the compound component. The interconnections among the internal components are not shown in the documentation of the interface of the compound component, but they are essential to the construction of the compound component. In our model, all communication between components is by shared variables. One component writes to a variable and other components read the same variable. Higher level communication mechanisms such as program calls and message transfer are always implemented using shared global variables.

2.3 A network of components

We view a component (as well as a compound component) as a black box with input/output variables. As with hardware, the value of an output at a time t is controlled by the inputs at time t or earlier. In other words, an output value can change without delay when an input value changes but components can have memory, i.e. the output can also depend on earlier input values.

If one component receives an output from another component as an input, the output and the input are viewed as connected together. The values of the connected input and output variables are always equal without delay. An output can be connected to many inputs, but to avoid any ambiguity about the value of an input, each input is only connected to one output. If it is desired to connect several outputs to an input it is necessary to have a multiplexer component that receives those inputs and delivers a single output value.

We call this collection of components and their interconnections a component network. Each component in a network is viewed as a hardware-like device in which an output value can change instantaneously when input values change and all components operate synchronously rather than in sequence. This is analogous to the concept of network of processes in [13].

Definition 2.3 A Network of Components (NC) is described by a quintuple $(\Delta, \Sigma, \Gamma, \Omega, \mathfrak{R})$ where:

- Δ: a set of components,
- Σ: the set of input variables of the network - variables whose values are determined by the environment outside the network,
- Ω: the set of output variables of the network - variables visible outside the network whose values are determined by the network,
- Γ: the set of interconnector variables whose values are determined within the network - and are not visible outside the network,
- $\mathfrak{R}$: a set of rules - the connection of input and output variables in the network. These rules are described below.

Denote a member of Σ by σ, a member of Γ by γ, and a member of Ω by ω.

There are three kinds of network rules in $\mathfrak{R}$ describing the interconnections of components and the associations between the components and the system. The following notations are defined:

1. $i \leftarrow \gamma \leftarrow o$ ($i \leftarrow \gamma \leftarrow o \in \mathfrak{R}$) indicates that an input i get values from output variable o of another component in the network, through the global interconnector variable γ , where i is a component input, o a component output and γ an interconnector.

It describes the connection relationship among inputs of one component and outputs of other components inside the network.

2. $i \leftarrow \sigma$ ($i \leftarrow \sigma \in \mathfrak{R}$) to indicate that the value of i is determined by the environment of network, where i is a component input and σ a system input.

It describes which system input should connect to which component input.

3. $o \rightarrow \omega$ ($o \rightarrow \omega \in \mathfrak{R}$) to indicate that the value of network output ω is determined by the component output o, where o is a component output and ω a system output variable.

It describes which component output should connect to network output.

In a mathematical sense, the interconnector variable γ is redundant, and the rule $i \leftarrow \gamma \leftarrow o$ can be writen as $i \leftarrow o$. However, from the documentation point of view, the interconnector variable is a place to mention the network role of a shared variable. The role of a shared variable for the writing component, any reading component and the network might all be different. For example, consider a distance sensor component with an output variable named 'distance'. If this sensor is mounted pointing upwards in a network, the interconnector used for its output could be named 'headroom'. For a component that reads the variable within a network, the name of the input variable might be quite different; such a component might perform a generic task (e.g, averaging) and not be restricted to distances. The input variable of this component might be named something like "in" or "raw".

A network is static; the connections do not change with time. The interconnection is delay-free; the values of connected input and output variables are always equal and can only change simultaneously, there is no delay. If there is delay in the system, it occurs inside a component.

2.4 The purpose of defining a network of components

After a system has been divided into components and a set of documents describing the externally visible behavior of the components has been prepared, and before starting further implementation, we should be able to check that the design is correct in the sense that if the implemented components are correct, we would get a system that behaves as required.

Our goal is to check whether the set of components' TFM descriptions (or any other type of document with the same information content) are consistent and whether they conform to a requirement specification of the system. A component network description as defined in Definition 2.3 is used to describe the interconnections among the components and the connections between the components and the software environment. Only with this information can we check whether or not the set of components can work together properly. Therefore the interconnection description of the components must be an integral part of a system documentation.

3 The completeness of a network of components

As described in the previous sections, within a network the components are interconnected by means of their interfaces. The completeness of a network is used to describe whether all necessary input variables are connected by output variables of other components or system inputs, and whether the values of an output variable could be calculated by one and only one component.

Definition 3.1 A network NC= $(\Delta, \Sigma, \Gamma, \Omega, \mathfrak{R})$ of components is said to be complete if and only if:

1. For each component and each component input i, there is one and only one $\gamma \in \Gamma$ such that $i \leftarrow \gamma$ ($i \leftarrow \gamma \leftarrow o \in \mathfrak{R}$, o is an output variable of another component) or one and only one $\sigma \in \Sigma$ such that $i \leftarrow \sigma \in \mathfrak{R}$.

It requires that each input to a component be determined by one and only one interconnector or network input. It guarantees that every assumption made by a component should be met by its environment.

2. For each interconnector γ in the network, there is one and only one component and one and only one output o such that $\gamma \leftarrow o$ ($i \leftarrow \gamma \leftarrow o \in \mathfrak{R}$, i is an input variable of another component).

It requires that each interconnector be connected to one and only one output of a single component in the network.

3. For each network output $\omega \in \Omega$, there is one component and one and only one output o such that $(o \rightarrow \omega \in \mathfrak{R})$.

It requires that each output that is required by the system must be calculated by one component.

Within a network each input and output variable of a component should have a unique name. If two components read from the same input variable, the input variable of each component will use different name and will be connected to the same variable. But this is not applicable to output variables since the network completeness restrictions do not allow shared output variables.

4 The consistency of a network of components

Consistency here means that the data types of interconnected variables are consistent and the network is stable thus lead to a well-defined behavior.

4.1 Data type consistency

The first step towards consistency between connected components is that data types in each network rule should be consistent. A network rule of the form $i \leftarrow \sigma$ says that variable i will receive values from variable σ. The set of possible values for the variable on the right must be a subset of the set of possible values for the variable on the left. The network could be possibly consistent only when data types of the variables in each network rule are consistent.

For the concepts of data types and type consistency, we use those definitions in [21] by replacing the term "mode" with "subtype" here to avoid possible misunderstanding, because in current and recent literature "mode" is mostly used with another meaning, and from the definition of mode in that paper we can see it is a definition of "subtype" in a more general way than many type systems now.

Definition 4.1 Types are classes of variables with certain stated properties. Subtypes are classes of variables whose data representation and access are identical. If a data type includes more than one subtype, it is an abstract type.

Subtypes are combined to abstract data types for variety of reasons such as to support the goals of abstraction, abbreviation, and code sharing without sacrificing type checking. Members of a subtype of a type may have more properties in common than the full type.

We define a type-subtype relation as:

Definition 4.2 The relation R_{ts} contains all pairs (A, B) of types where A is a subtype of B.

This relation is reflexive and transitive. Denote the data type of a variable v as $\mathtt{Type}(v)$, it returns the data type of v.

Definition 4.3 The data type of the variables in a network are consistent only when:

1. For $i \leftarrow \gamma \leftarrow o \in \mathfrak{R} \Rightarrow (\mathtt{Type}(i), \mathtt{Type}(\gamma)) \in R_{ts}$ and $(\mathtt{Type}(\gamma), \mathtt{Type}(o)) \in R_{ts}$
2. For $i \leftarrow \sigma \in \mathfrak{R} \Rightarrow (\mathtt{Type}(i), \mathtt{Type}(\sigma)) \in R_{ts}$
3. For $o \rightarrow \omega \in \mathfrak{R} \Rightarrow (\mathtt{Type}(\omega), \mathtt{Type}(o)) \in R_{ts}$

4.2 Delay-free loops in the network

For any two components, our definition of a network allows communication in both directions. Consider a pair of components that is connected such that component C_1 writes to variable γ_1 and reads γ_2, whereas component C_2 reads variable γ_1 and writes to γ_2: if the output of one of the components depends only on earlier values of its input, the behavior of such networks will be well defined. However, if both output values depend on current input values (meaning that there are no delays in

either component) the network may not have a unique stable state and the behavior could be undefined. In other words, if there is a loop in a network, at least one of the components within the loop should have delay so that not all the interconnector variables are changed simultaneously.

This means that the component description must contain sufficient information to tell us whether or not there is a delay between an input and an output. In TFM this is easily determined by looking at the variables that appear in the expression defining the output function.

Definition 4.4 A dependency description, D, is a set of rules of the form $o \leftarrow \cdot i$ describing the fact that the value of output variable o at time t may depend on the value of input variable i at the same time. The D rules can be derived directly from a TFM document but should be derivable from any other form of functional interface documentation.

There are three kinds of loops within two components: a) delay-free loop, where the value of γ_1 depends on the current value of γ_2 and vice versa, b) delay in one component, where only one of the variables depends on the current value of another, and c) delay in both components, where both variables have no dependencies on the current value of another.

With the D-rules a delay-free loop is defined as:

Definition 4.5 A delay-free loop between two components C_1 and C_2 is the case when $i_2 \leftarrow \gamma_1 \leftarrow o_1 \in \mathfrak{R}$ and $i_1 \leftarrow \gamma_2 \leftarrow o_2 \in \mathfrak{R}$, $o_1 \leftarrow \cdot i_1 \in D_1$ and $o_2 \leftarrow \cdot i_2 \in D_2$. In such a situation the calculation of the output of C_1 that is sent to C_2 at time t depends on an input value that is receiving value from C_2 at the same time, and vice versa.

Definition 4.5 characterizes a special case of delay-free loops. If there are n components in a network, such kind of loops may go through k $(2 \leq k \leq n)$ components and the definition can be easily augmented to describe loops in more than two components.

Definition 4.6 There is a delay-free loop among k components in the network if there exists a sequence of variables such that $i_2 \leftarrow \gamma_1 \leftarrow o_1 \in \mathfrak{R}$, $i_3 \leftarrow \gamma_2 \leftarrow o_2 \in \mathfrak{R}$, $\ldots$, $i_1 \leftarrow \gamma_k \leftarrow o_k \in \mathfrak{R}$, and for all $i \in \{1, \ldots, k\}$, $o_i \leftarrow \cdot i_i \in D_i$.

4.3 Why rule out delay-free loops?

The presence of a delay-free loop indicates that the network may be ill-formed and may have undefined behavior in some cases. If such a loop is found, we consider it a design error. Nevertheless, this restriction is not yet justified. The P-T (Parnas-Trancon) Conjecture stated below, and not yet proven, is the motivation of ruling out this case from our consideration.

P-T Conjecture: For a delay-free loop, only the following two cases could happen:

(i) It is unstable or has more than one stable state, none of which is clearly preferable over the others.

(ii) It is stable with exactly one solution, and there is an equivalent network without such a loop.

The meaning of "solution" becomes obvious if the case is formulated as mutually recursive function equations:

$$o_1(T) = f(p(T), o_2(T)), \qquad o_2(T) = g(p(T), o_1(T))$$

where o_1 and o_2 are variables of two components, $o_1(T)$ and $o_2(T)$ are the respective values of these variables at the end of a trace T, $p(T)$ is the trace containing all but the last event of T, and f and g are trace functions defined in a TFM specification/description.

By "equivalent network" we mean a network that always produces the same output when given the same input values.

Proving or disproving the conjecture is future work.

Definition 4.7 A network of components is said to be consistent if and only if:

(i) The data type in each network rule is consistent as defined in Definition 4.3; and

(ii) There is no delay-free loop in the network.

5 Summary and future work based on the network description

We have defined a network of components and discussed the rules for checking when the network is complete and consistent, which are:

(i) Each output variable can be an output variable of exactly one component, while input variables could be shared.

(ii) In every network rule $i \leftarrow \gamma \leftarrow o$, the data type of the three variables must match as defined in Definition 4.3.

(iii) In every network rule $i \leftarrow \gamma \leftarrow o$, γ gets the value from o, and i gets its value from γ without delay.

(iv) If there is a loop in reading and writing between two components, there should be delay in the loop.

If a network is complete and consistent as discussed in the above sections, the connected components could work together. But whether the components could work together correctly to finish the required task is not yet assured. Checking the correctness of a TFM description of network behavior is the aim of the next stage of our research.

A network behaves correctly if all the component TFM descriptions and the connection description together capture the required software behavior. In future work we will find ways to confirm that the behavior of the network satisfies the requirements.

References

[1] R. Allen, D. Garlan, A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, July 1997.

[2] R. Bharadwaj and C.L. Heitmeyer, Verifying SCR requirements specifications using state exploration, *Proc. 1st ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.
[3] W. Bartussek and D.L. Parnas, *Using Assertions about Traces to Write Abstract Specifications for Software Modules*, UNC Report No. TR77-012, Dec. 1977.
[4] R. Baber, D.L. Parnas, S. Vilkomir, P. Harrison, T. O'Connor, Disciplined Methods of Software Specifications: A Case Study, *Proceedings of the International Conference on Information Technology Coding and Computing* (ITCC 2005), Las Vegas, NV, USA, IEEE Computer Society, April 4-6, 2005.
[5] E.W. Dijkstra, The Structure of "THE"-Multiprogramming System, *Communications of the ACM*, Vol.11, No.5, pp.341-346, 1968.
[6] F. DeRemer and H. Kron, Programming-in-the-Large Versus Programming-in-the-Small, *IEEE Trans. Software Engineering*, vol.2, pp.321-327, 1976.
[7] D. Garlan and M. Shaw, An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, Vol.I, 1993.
[8] C. Heitmeyer and R. Jeffords, B. Labaw, Automated Consistency Checking of Requirements Specifications, *ACM Trans. Software Engineering and Methodology*, Vol.5. No.3, pp.231-261, 1996.
[9] M. P. E. Heimdahl and N. G. Leveson, Completeness and Consistency in Hiearchical State-Based Requirements, *IEEE Trans. Software Engineering*, Vol.22, No.6, pp.363-377, 1996.
[10] C. Heitmeyer, B. Labaw and D. Kiskis, Automated Consistency Checking of SCR-Style Requirements Specifications, *ACM Transactions on Software Engineering and Methodology*, Vol. 5, No. 3, pp.231-261, July 1996,
[11] M. Jin, *Table Checking Tool*, Master Thesis, McMaster University, Hamilton, Canada, 2000.
[12] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, *IEEE Transactions on Software Engineering*, Volume 26, Issue 1, pp.70-93, Jan. 2000
[13] D. L. Parnas, On Simulating Networks of Parallel Processes in Which Simultaneous Events May Occur, *Communications of the ACM*, vol.12, no.9, pp.519-531, 1969.
[14] D.L. Parnas, A Technique for Software Module Specification with Examples, *Communications of the ACM*, Vol.15, No. 5, pp.330-336 May 1972.
[15] D.L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, vol.15, no.12, pp.1053-1058, 1972.
[16] D. L. Parnas The Use of Precise Specification in the Development of Software *Proc. IFIP Congress'77*, North Holland Publishing Company, pp.861-867, 1977.
[17] D. L. Parnas, Some theorems we should prove, *Higher Order Logic Theorem Proving and its Applications* (6th International Workshop HUG'93), J. J. Joyce and C.-J. H. Seger, ed., pp.155-162, Vancouver, Canada, August 1993.
[18] D. L. Parnas and M. Dragomiroiu, Component Interface Documentation - Using the Trace Function Method (TFM), SQRL paper Aug. 2006 version.
[19] D.L. Parnas and J. Madey, Functional Documents for Computer Systems, *Science of Computer Programming* (Elesevier) 1995.
[20] R. Prieto-Diaz, J. M. Neighbors, Module Interconnection Languages, *The Journal of Systems and Software 6*, pp.307-334, 1986
[21] D.L. Parnas, J.E. Shore, and D. Weiss, Abstract Types Defined as Classes of Variables, *Proc. Conf. on Data: Abstraction, Definition, and Structure*, Salt Lake City, March 1976, pp.22-24. Reprinted in NRL Memorandum Report 7998, pp.1-10, April 1976.
[22] D. L. Parnas and Y. Wang, *The Trace Assertion Method of Module Interface Specification*, Technical Report, 89-261, TRIO, Queen's University, October 1989.
[23] C. Quinn, S.A. Vilkomir, D.L. Parnas, S. Kostic, Specification of Software Component Requirements Using the Trace Function Method. *Proceeding of the International Conference on Software Engineering Advances* (ICSEA 2006), Tahiti, French Polynesia, October 29 - November 1, 2006.
[24] W. F. Tichy, Software Development Control Based on Module Interconnection, *IEEE Proceedings of the 4th international conference on Software engineering*,1979.
[25] Y. Wang, *Specifying and simulating the externally observable behaviour of modules*, Doctoral Thesis, McMaster University, Hamilton, Canada, 1994.
[26] Ou Wei, *Preliminary Requirements Checking Tool*, Master Thesis, McMaster University, Hamilton, Canada, 2001.
[27] J. Zhang, Search Techniques for Testing Formal Specifications, *Proc. 10th Int. Conf. Software Engineering and Knowledge Engineering*, California, USA, 1998.

Replace this file with `prentcsmacro.sty` for your meeting, or with `entcsmacro.sty` for your meeting. Both can be found at the ENTCS Macro Home Page.

From Bidirectional Model Transformation to Model Synchronization

Yingfei Xiong[1]

Department of Mathematical Informatics
The University of Tokyo, Tokyo, Japan

Song Hui[2]

Key Laboratory of High Confidence Software Technologies (Peking University)
Ministry of Education, Beijing, China

Zhenjiang Hu[3]

GRACE Center
National Institute of Informatics, Tokyo, Japan

Masato Takeichi[4]

Department of Mathematical Informatics
The University of Tokyo, Tokyo, Japan

Abstract

In model-driven engineering, it is common that there are several related models co-existing. When one model is updated or several models are updated at the same time, we need to propagate the updates across all models to make them consistent. This process is called synchronization. Bidirectional model transformation partially supports the synchronization of two models by updating one model according to the other models. However, it does not work when the two models are modified at the same time.
In this work we propose a new algorithm that wraps any bidirectional transformation into a synchronizer, and this synchronizer allows simultaneous updates on the two models. We propose a general algebraic framework for model synchronization, and prove that our algorithm can ensure the synchronization properties if the bidirectional transformation satisfies the correctness property and the hippocraticness property [7].

Keywords: bidirectional transformation, model synchronization, model transformation

[1] Email: xiong@ipl.t.u-tokyo.ac.jp
[2] Email: songhui06@sei.pku.edu.cn
[3] Email: hu@nii.ac.jp
[4] Email: takeichi@mist.i.u-tokyo.ac.jp

©2009 Published by Elsevier Science B. V.

1 Introduction

One central activity of model-driven software development is to transform high-level models into low-level models through model transformations. In an ideal situation, the target model is always obtained from the source model and never need to be modified. However, in reality, developers often need to modify the target model directly. In such cases, we need to reflect the updates on the target models back to the source models.

Bidirectional model transformation solves this maintenance problem by providing bidirectional model transformation languages, which describe the relation between two models symmetrically. Programs in these languages are able to not only transform models from one format into the other, but also update the other model automatically when one model is updated by users. Typical bidirectional model transformation languages include QVT [5] and TGG [4].

Perdita Stevens [7] formalizes bidirectional model transformation as two functions. If M and N are meta-models and $R \subseteq M \times N$ is the consistency relation to be established on the models. A bidirectional transformation consists of the following two functions:

$$\overrightarrow{R} : M \times N \to N$$
$$\overleftarrow{R} : M \times N \to M$$

Given a pair of models $(m, n) \in M \times N$, the function $\overrightarrow{R}$ changes n to be consistent to m. Similarly, $\overleftarrow{R}$ changes m in accordance with n.

However, in some cases the the model m and n may both be updated before bidirectional transformation can be applied. For example, a designer is working on the design model and a programmer is working on the implementation model at the same time. Applying the transformation of any direction will result in the loss of updates on the target side.

To solve this problem, we need a synchronizer to propagate the updates on each model to the other model at the same time. In this paper we consider such a synchronizer as a partial function

$$sync : R \times (M \times N) \to R$$

that takes two original models in the consistency relation R, two updated models and produces two synchronized models. The output model should be close to the original models, and also contains the updates in the updated models and the updates propagated from the other sides. The function is partial because sometimes the updates on the two models may conflict and cannot be synchronized.

Given the large number of available bidirectional model transformation languages, there are relative few ready-to-use synchronization languages. So one natural idea is to use bidirectional model transformation to support model synchronization. In this paper we carry out theoretical studies of how bidirectional model transformation can be used to support model synchronization. The main contributions of this paper can be summarized as follows:

- We extend our previous algebraic framework for model synchronization [8] to general cases. We consider the symmetrical cases where no model is necessarily

an abstraction of the other model and open door to free choice of updates.

- We propose an algorithm that wraps any bidirectional model transformation into a synchronizer, with the help of a three-way merger. We also discuss basic conflict-resolving support in the synchronizer.
- We prove that, for any bidirectional transformation satisfying the correctness and hippocraticness properties [7], the synchronizer satisfies the stability, preservation and consistency properties [8], ensuring a correct and predictable synchronization behavior.

This paper is organized as follows. Section 2 introduces our algebraic framework of model synchronization, including three properties to characterize the behavior of synchronization. Section 3 introduces the bidirectional model transformation properties introduced by Stevens [7]. Based on these properties, Section 4 introduces our algorithm and prove that bidirectional model transformation properties lead to model synchronization properties. Section 5 introduces our basic conflict-resolving strategy. Finally, Section 6 discusses two pieces of related work.

2 Properties of Model Synchronization

We have seen the basic definition of a synchronizer: it takes two original models, two updated models and produces two synchronized models. However, this definition only characterize the input and output types of the synchronizer, and does not say much about the synchronization behavior. In this section we propose several properties to characterize the behavior of the synchronizer.

As the first step of charactering the behavior, let us define the updates on the models. In our definition, the synchronizer only takes models and produces new models, and one may ask: why do we need to consider updates? This is because we need to detect updates and merge simultaneous updates in synchronization. If we consider different sets of updates, the synchronization may lead to different results.

For example, let us consider the meta model M as a power set of some alphabet set Σ. Suppose two users made two different updates on one model, respectively, and their updated results are as follows.

the original model	$M_0 : \{a, b, c\}$
the first updated result	$M_1 : \{a, d, c\}$
the second updated result	$M_2 : \{a, e, c\}$

If we consider M_1 is created by replacing b by d, and M_2 is created by replacing b by e, the two updates will conflict. However, if we consider M_1 is created by deleting b and adding d, while M_1 is created by deleting b and adding e, this two updates are compatible and we can merge them as one model: $\{a, d, e, c\}$.

From these different results we can see that the synchronization behavior depends on what updates we choose. To clear characterize the behavior, we need to first be clear about which set of updates we will consider during the synchronization. First we give the definition of update: An *update* u defined on some meta-model M is an idempotent function $u \in M \to M$. We consider only the idempotent function because idempotence allows us to tell whether the update has been preserved in

a model. If we apply an update to a model and the model remain constant, the update has been preserved in the model.

Example 2.1 The meta model M is a power set of some alphabet set Σ. Suppose we have the following functions:

- $add[\![a]\!](A) = A \cup \{a\}$
- $remove[\![a]\!](A) = A \backslash \{a\}$
- $replace[\![a, b]\!](A) = \begin{cases} A\backslash\{a\} \cup \{b\} & a \in A \\ A & \text{otherwise} \end{cases}$

Then for any $a, b \in \Sigma$, we have $add[\![a]\!]$, $remove[\![a]\!]$ and $replace[\![a, b]\!]$ are updates.

After we define updates as functions, the relationship between updates can be defined through function composition. Two updates u_1, u_2 *conflict* iff $u_1 \circ u_2 \neq u_2 \circ u_1$. We write $u_1 \ominus u_2$ if u_1 and u_2 do not conflict.

Corollary 2.2 *$\ominus$ is commutative.*

Proof. By the definition. □

Corollary 2.3 *If $a \ominus b$, we have that $a \circ b$ is an update.*

Proof. Because $a \circ b = b \circ a$, we have $(a \circ b) \circ (a \circ b) = (a \circ a) \circ (b \circ b) = a \circ b$. □

Corollary 2.4 *If $b \circ c$ is an update and $a \ominus b$, $a \ominus c$, we have $a \ominus (b \circ c)$.*

Proof. Because $a \ominus b$, we have $a \circ b = b \circ a$. Putting together $a \ominus c$, we have $a \circ (b \circ c) = b \circ a \circ c = b \circ (a \circ c) = b \circ c \circ a$. □

Another relation we consider is whether an update is included in another update. An update u_1 is a *sub update* of another update u_2 iff $u_1 \circ u_2 = u_2 \circ u_1 = u_2$, denoted as $u_1 \sqsubseteq u_2$.

Corollary 2.5 *$\sqsubseteq$ is a partial order over any set of updates.*

Proof. By definitions. □

Proof. We need to show $\sqsubseteq$ is reflexive, antisymmetric and transitive.

Reflexive $a \circ a = a$

Antisymmetry If $a \sqsubseteq b$ and $b \sqsubseteq a$, we have $a \circ b = a$ and $a \circ b = b$, and then we have $a = b$.

Transitivity If $a \sqsubseteq b$ and $b \sqsubseteq c$, we have $a \circ b = b \circ a = b$ and $b \circ c = c \circ b = c$, then we have $a \circ c = a \circ (b \circ c) = (a \circ b) \circ c = b \circ c = c$. Similarly ,we can have $c \circ a = c$. □

Corollary 2.6
$\forall a, b \in \Sigma : a \neq b \Rightarrow add[\![a]\!] \ominus add[\![b]\!]$
$\forall a, b \in \Sigma : a \neq b \Rightarrow remove[\![a]\!] \ominus remove[\![b]\!]$

$\forall a, b \in \Sigma : a \neq b \Rightarrow add[\![a]\!] \ominus remove[\![b]\!]$

As we have discussed, to clearly characterize the synchronization behavior of a synchronizer, we need to know what kinds of updates can be applied to a model. We define the updates that can be applied to models in M through a *proper update set* U_M. A proper update set U_M defined on a meta model M is a set of updates that satisfies:

- U_M is closed on composition,
- the identity function $id \in U_M$, and
- for any $m, n \in M$, the set $\{u \in U_M | u(m) = n\}$ has a least element.

The first condition requires the property update set to be complete, so that we can freely apply a sequence of updates in the set without worrying whether we are applying a "proper update". The second condition allows us to keep the model unmodified. The third condition implies two things: 1) any model can be applied to any other model, and 2) given two models, we can always find a unique update that is least among all updates.

If u is the least element in the set $\{u \in U_M | u(m) = n\}$ for any $m, n \in M$, we say u is the least update from m to m'.

To give an example of proper update set, let us define a function which construct a set of functions by composing another set of function with a predefined set: $compose[\![F]\!](H) = \{f \circ h \mid \forall f \in F, h \in H\}$

Lemma 2.7
$\forall a \in \Sigma : add[\![a]\!] \circ remove[\![a]\!] = add[\![a]\!]$
$\forall a \in \Sigma : remove[\![a]\!] \circ add[\![a]\!] = remove[\![a]\!]$

Example 2.8 Let
$B_1 = \{add[\![a]\!] \mid \forall a \in \Sigma\} \cup \{remove[\![a]\!] \mid \forall a \in \Sigma\}$,
$M_1 = \bigcup_{n=0}^{\infty}(compose[\![B_1]\!])^n(\{id\})$,
we have M_1 is a proper update set.

Proof. First, every element in M_1 is an update. Every element in M_1 can be written as a sequence of composition $oper[\![a_0]\!] \circ oper[\![a_1]\!] \ldots oper[\![a_n]\!]$ where $oper = add$ or $remove$ and $a_i \neq a_j$ for any $i \neq j$. This can be proved by mathematical induction. Furhter because of Corollary 2.7, Corollary 2.3 and Corollary 2.4, every element is an update.

Second, similarly, we have M_1 is closed on composition.

Third, by definition, id is in M_1.

Forth, consider two element m_0 and m_1 in M. Let $set_1 = \{add[\![a]\!] \mid a \in m_0 \wedge a \notin m_1\}$ and $set_2 = \{remove[\![a]\!] \mid a \in m_1 \wedge a \notin m_0\}$. The least update from m_0 to m_1 is a composition of all element in set_1 and set_2. We can easily prove this is a sub update of any other updates. □

Example 2.9 Let
$B_2 = B_1 \cup \{replace[\![a, b]\!] \mid \forall a, b \in \Sigma\}$,

$M_2 = \bigcup_{n=0}^{\infty}(compose[\![B_2]\!])^n(\{id\})$,
we have M_2 is not a proper update set.

Proof. Given two sets $m_1 = \{a, b\}$ and $m_2 = \{a, c\}$, both $add[\![c]\!] \circ remove[\![b]\!]$ and $replace[\![b, c]\!]$ can update m_1 to m_2, but none is a sub update of the other. □

With updates clearly defined, we are ready to move to define the properties. We consider stability, preservation and consistency defined in [8] and leave the composability property, which is arguably too strong. The three properties are previously defined in the case where the target model is an abstraction of the source model, here we adapt the definitions to symmetrical cases.

Stability says that if no model is updated, the synchronizer should update no model.

Property 1 (Stability)
$R(m, n) \Rightarrow sync(m, n, m, n) = (m, n)$

Preservation requires user updates should be preserved during synchronization. In other words, when users modify a data item to some specific valuess, the synchronizer should not modify the data item to any other value.

Property 2 (Preservation)
If $sync(m, n, m', n') = (m'', n'')$, u_m *is the least update from* m *to* m', *we have* $u_m(m'') = m''$
If $sync(m, n, m', n') = (m'', n'')$, u_n *is the least update from* n *to* n', *we have* $u_n(n'') = n''$

Consistency requires the synchronizer to produce consistent result.

Property 3 (Consistency)
$sync(m, n, m', n')$ *is defined* $\Rightarrow R(sync(m, n, m', n'))$

3 Properties of Bidirectional Model Transformation

Perdita Stevens [7] also proposes three properties to ensure a predictable behavior of bidirectional model transformations. Two of the properties are correctness and hippocraticness. The third property, undoability, is also arguably too strong, and we do not consider it here.

Property 4 (Correctness)
$\forall m \in M, n \in N \qquad R(m, \overrightarrow{R}(m, n))$
$\forall m \in M, n \in N \qquad R(\overleftarrow{R}(m, n), n)$

Property 5 (Hippocraticness)
$R(m, n) \Rightarrow \overrightarrow{R}(m, n) = n$
$R(m, n) \Rightarrow \overleftarrow{R}(m, n) = m$

4 Algorithm

The basic idea of the algorithm is to first convert the model in one side to the other side using bidirectional transformation, then use a three-way merger [3] to reconcile

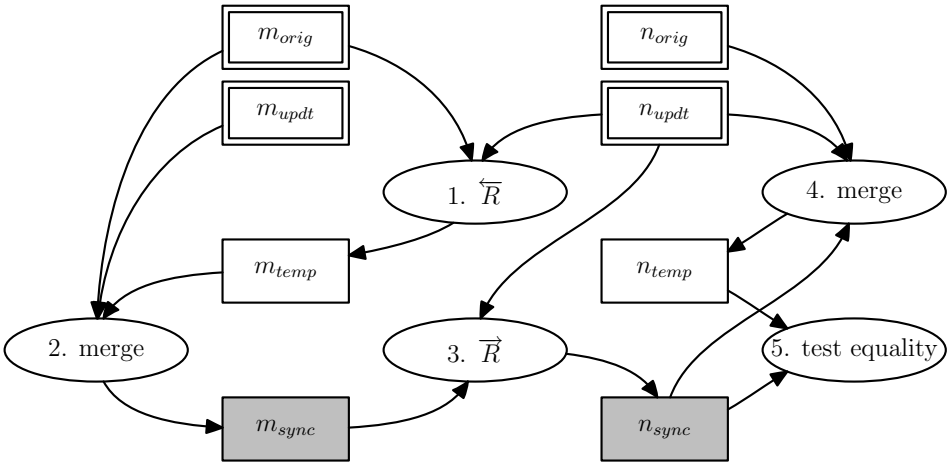


Fig. 1. The Synchronization Algorithm

the updates, and transform back using the opposite transformation. The detailed algorithm is shown in Figure 1.

Initially, we have the original models m_{orig}, n_{orig} and the updated models m_{updt}, n_{updt}. First we use $\overleftarrow{R}$ to propagate the updates on n_{updt} to m_{orig} and we get m_{temp}. Then we invoke a three-way merger to merge m_{orig}, m_{updt} and m_{temp}.

A three-way merger is a partial function $merge \in M \times M \times M \to M$ that takes a reference model m_o and two updated models m_a and m_b diverged from m_o, and produced a new model m'_o where the updates in m_a and m_b are reconciled. Suppose u_a is the least update from m_o to m_a and u_b is the least update from m_o to m_b, the $merge$ function will ensure the following:

- $merge(m_o, m_a, m_b)$ is not defined iff u_a and u_b conflict.
- $merge(m_o, m_a, m_b) = m'_o \Rightarrow (u_a \circ u_b)(m_o) = m'_o$.

Back to our algorithm, here m_{updt} contains the update on m_{orig} and m_{temp} contains the update transformed from n_{orig}. After we merge them using m_{orig} as a reference model, we can get m_{sync} that contains updates from both sides.

When we have a synchronized model m_{sync} on M side, we can perform $\overrightarrow{R}$ to get a synchronized model n_{sync} on N side, and the n_{sync} should contains updates from both side.

Now we have two synchronized models where the updates are propagated. It looks that we have performed enough steps to finish the algorithm. However, the above steps is not always able to detect all conflicts, and may lead to violation of preservation due to the heterogeneousness of the two models

To see how this can happen, let us consider the following example. Suppose M contains two constants $\{a, b\}$ and N contains two constants $\{x, y\}$. The consistency relation between them is

$$\left\{ \begin{array}{l} (a, x) \\ (a, y) \\ (b, x) \end{array} \right\},$$

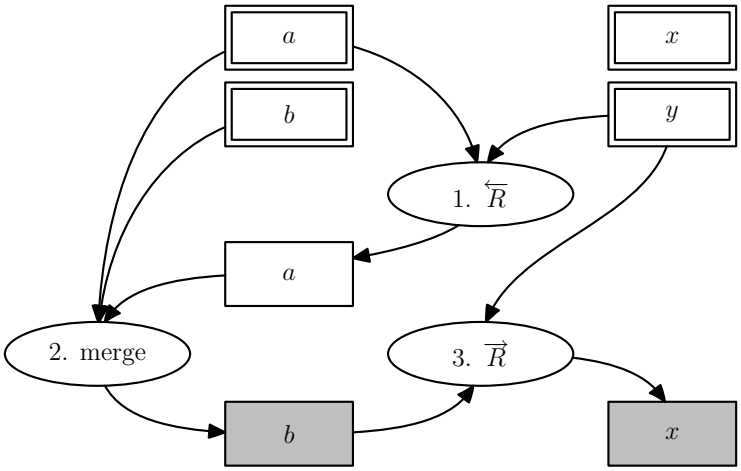


Fig. 2. An Example Violating Preservation

That is, a is related to x and y while x is also related to b. Suppose initially the two models are $m_{orig} = a$ and $n_{orig} = x$, and then m_{orig} is updated to b while n_{orig} is updated to y.

The process of this computation is shown in Figure 2. After computation, the algorithm produces b on the M side and x on the N side. However, if we check the update on the N side, we will find that x is updated to y and this updated is not preserved in the synchronized model. The property of preservation is violated.

The violation is caused by the asymmetry of M and N. Both x and y in N are related to the same element a in M. When n_{updt} is transformed to the M side, the update is not recognizable by the state-based three-way merger.

To capture such conflict, we add an additional preservation check at the end of the synchronization. As shown in the 4th and the 5th steps in Figure1. We first merge n_{updt} and n_{sync} with n_{orig} as a reference, and then compare whether the merged model n_{temp} is equal to n_{sync}. If the preserve property is satisfied, the two model should be equal, otherwise the algorithm will report an error message indicating there are conflicts.

Theorem 4.1 *If the bidirectional transformation $(\overrightarrow{R}, \overleftarrow{R})$ satisfies correctness and hippocraticness, we can ensure that the synchronization algorithm satisfy stability, consistency and preservation.*

Proof. Stability If we have $m_{orig} = m_{updt}$ and $n_{orig} = n_{updt}$, then we have $R(m_{orig}, n_{updt})$. Because of hippocraticness, $m_{temp} = \overleftarrow{R}(m_{orig}, n_{updt}) = m_{orig}$. Because the least update from m_{orig} to m_{updt} and to m_{temp} are both id, $merge$ will produce the same model, that is $m_{sync} = m_{orig}$. Similarly, $n_{sync} = n_{orig}$ and the preservation check always passes successfully.

Preservation On the M side, suppose u_m is the least update from m_{orig} to m_{updt}, because $merge(m_{orig}, m_{updt}, m_{temp}) = m_{sync}$, we have $u_m(m_{sync}) = m_{sync}$. Similarly, suppose u_n is the least update from n_{orig} to n_{updt}, because $merge(n_{orig}, n_{updt}, n_{sync}) = n_{temp} = n_{sync}$, we have $u_n(n_{sync}) = n_{sync}$.

Consistency Because $\overrightarrow{R}(m_{sync}, n_{updt}) = n_{sync}$, we have $R(m_{sync}, n_{sync})$. □

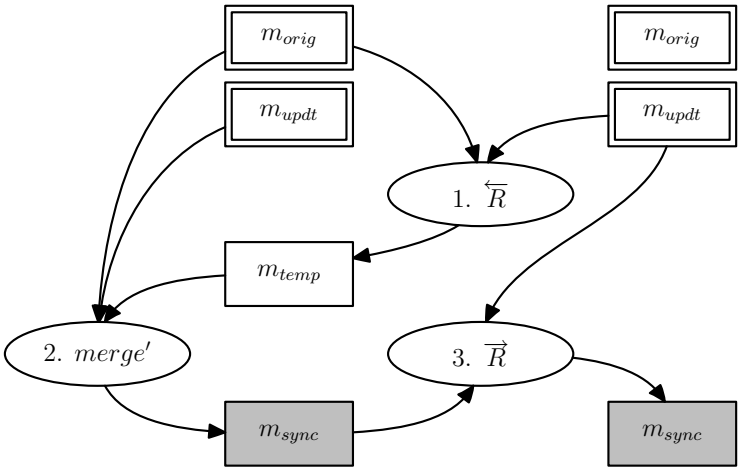


Fig. 3. The Synchronization Algorithm for Conflict Resolving

5 Conflict Resolving

One important issue in synchronization is how to resolve conflicts in the two updated models, automatically or with user intervention. A full discuss of conflict resolving relates to conflict presentation and interaction, which is beyond the scope of the paper. In this section we consider a simple automatic resolving strategy: overwriting all conflicting updates on one model with the updates on the other model.

Let us first consider the case where the updates in M take priority. Because bidirectional transformation describe the transformation symmetrically, we can just swap M and N when we need N to take priority.

Because of the updates in N may be overwritten by the updates in M, we need to loosen the preservation property to allow loss of updates on the N side. The loose preservation property only requires to preserve updates on the M side, as the following.

Property 6 (Loose Preservation for Conflict Resolving)
If u_m is the least update from m to m' and $sync(m, n, m', n') = (m'', n'')$, we have $u_m(m'') = m''$

Furthermore, we need an extended merge function $merge'$ which deals with conflicting updates and gives priority to the first model.

- $merge' : M \times M \times M \to M$ is a total function.
- $merge'(m_o, m_a, m_b) = m'_o \Rightarrow (u_a \circ u_b)(m_o) = m'_o$.

The algorithm for the updated model is shown in Figure 3. This algorithm is similar to the original one, except that we use $merge'$ instead of $merge$ and we do not post-check preservation. We can similarly prove that the algorithm ensure stability, consistency and the loose preservation if the bidirectional transformation satisfies correctness and hippocraticness.

6 Related Work

Pierce and et al. [6] propose the Harmony framework, which also addresses the issue of supporting synchronization from bidirectional transformations. Compare to our work, Harmony emphasizes on totality, but requires users to design middle model and two transformation which relates the middle model to the original two models respectively. In this way Harmony can achieve totality, but it requires more programming work to design the middle model and code the two transformations.

Antkiewicz and Czarnecki discuss various design decisions of synchronizers in their work[1]. Their work classifies synchronizers into different types using different design decisions. Use their classification, our synchronization algorithm can be classified as "bidirectional, non-incremental, and many-to-many synchronizer using artifact translation, homogeneous artifact comparison and reconciliation with choice".

7 Conclusion

In this paper we propose an approach that wraps a bidirectional transformation program into a synchronizer for simultaneous updates. Our approach is *general*, in the sense that it allows any bidirectional transformation, and *predictable*, satisfying the model synchronization properties: consistency, stability and preservation.

Our approach is built upon idempotent updates. However, in the real world many updates cannot easily be presented as idempotent functions. For example, "inserting the item a into a list at index 2" is not an idempotent function. In our future work we plan to adopt more general definitions of updates (e.g., considering updates as arrows in a graph [2]), and extend our synchronization framework to more general cases.

References

[1] Antkiewicz, M. and K. Czarnecki, *Design space of heterogeneous synchronization*, in: *Proc. 2nd GTTSE*, 2007, pp. 3–46.

[2] Diskin, Z., *Algebraic models for bidirectional model synchronization*, in: *Proc. 11th MoDELS*, 2008, pp. 21–36.

[3] Khanna, S., K. Kunal and B. C. Pierce, *A formal investigation of diff3*, in: Arvind and Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 2007.

[4] Kindler, E. and R. Wagner, *Triple graph grammars: Concepts, extensions, implementations, and application scenarios*, Technical Report tr-ri-07-284, University of Paderborn (2007).

[5] Object Management Group, *MOF QVT final adopted specification*, http://www.omg.org/docs/ptc/05-11-01.pdf (2005).

[6] Pierce, B. C., A. Schmitt and M. B. Greenwald, *Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data*, Technical Report MS-CIS-03-42, University of Pennsylvania (2003).

[7] Stevens, P., *Bidirectional model transformations in QVT: Semantic issues and open questions*, in: *Proc. 10th MoDELS*, 2007, pp. 1–15.

[8] Xiong, Y., D. Liu, Z. Hu, H. Zhao, M. Takeichi and H. Mei, *Towards automatic model synchronization from model transformations*, in: *Proc. 22nd ASE*, 2007, pp. 164–173.

Functional Programming, Workflow and Cloud

Yike Guo

Imperial College London The United Kingdom

Abstract

Functional programming has been a subject of main stream computer science for many years. In spite of its huge impact on the development of programming language theory and practice, software engineering and even computer architecture, it is fair to say that functional programming is still mainly a principle rather than a particle tool for daily software development. However, I would like to argue that this situation may be changed in the new era of service oriented computing. In last few years, workflow has been an active research subject in the community of Grid computing. Workflow, which can be understood as a graphical functional programming language, provides a practical means for building computational services by composing existing services. Some practical workflow systems have been build and widely adopted in industry. This development has proven that the functional programming is playing an increasingly important role in service oriented computing. In this talk, I would like to present a new view of functional programming in the context of Cloud Computing. By presenting the technical challenges and the methodology of building Cloud application, I will present a functional framework for programming the Cloud by mashing up Cloud services.

On the Computation of Quotients and Factors of Regular Languages

Mircea Marin[1] and Temur Kutsia[2]

[1] Graduate School of Systems and Information Engineering
University of Tsukuba
Tsukuba 305-8573, Japan
[2] Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria

Abstract. Quotients and factors of regular languages are important notions in the design of computational procedures in the algebra of regular expressions and the analysis of their logical properties. We present algorithms for the computation of quotients and factors of languages specified by regular expressions.

1 Introduction

In [3], Brzozowski introduced the notion of word derivative and proposed an elegant algorithm to turn a regular expression into a deterministic finite automaton (DFA). His insight was that the word derivatives of a regular expression serve as states of the corresponding DFA. Antimirov [1] went a step further by introducing the notion of partial word derivative and observing that partial word derivatives of regular expressions serve as states of a corresponding nondeterministic finite automaton (NFA) with a relatively small number of states. He also noticed that computations based on partial word derivatives can improve the efficiency of Brzozowski's algorithm of translating regular expressions into finite automata. A much broader analysis of the role of word derivatives in the algebra of regular expressions was carried out by Conway [4]. His notion of left (resp. right) derivate of a language coincides with the current notion of language left quotient $F^{-1}E$ (resp. right quotient EF^{-1}), whereas his notion of right factor coincides with the recently proposed notion of product derivative [7]. Both quotients and factors are natural generalizations of the notion of word derivative, but in slightly different directions:

- The left quotient of E with respect to a language F is the *union* of all word derivatives of E with respect to a word from F,
- The right factor of E with respect to a language F, a.k.a. the product derivative of E with respect to F in [7], is the *intersection* of all word derivatives of E with respect to a word from F.

Quotients and factors of regular languages are relevant in the design of computational procedures in the algebra of regular expressions and the analysis of its

logical properties [4]. For example, factors of regular languages can be used in the computation of maximal solutions of formulas $P \leq R$ where R is a regular expression over an alphabet $\mathcal{A}$, P is a regular expression over an alphabet $\mathcal{A} \cup \mathcal{X}$, $\mathcal{X}$ is a countable set of variables, and "$\leq$" denotes language inclusion. This kind of generalized matching problem has finitely many maximal solutions which can be computed by using the factor matrix of R [4, Ch. 6].

Algorithms for the computation of quotient are well known for representations of regular languages by finite automata [5, Thm. 3.6], whereas algorithms for the computation of factors are more recent developments. In [7], the right factor is obtained from the computation of the greatest fixed point of a continuous operator which renders the result as a finite intersection of languages represented by regular expressions.

Our algorithms for the computation of quotient and factors manipulate languages represented by regular expressions, and compute a representation of the result by a regular expression. We identify the notion of *system of characteristic equations* of a regular language, show that such a system has a unique solution, that the solution consists of regular languages, and that there is a straightforward way to compute a representation of the solution by regular expressions. The main insights of our algorithms are:

- We can compute a system of characteristic equations for every regular language. This computation can be carried out in the differential calculus of Antimirov [1].
- We can compute systems of characteristic equations for the quotient and factors of regular languages E and F from systems of characteristic equations for E and F.

In this way, we reduce the computation of quotient and factors, to the computation and solving of systems of characteristic equations.

The paper is structured as follows. Section 2 presents basic notions and known theoretical results from the calculus of regular expressions. Section 3 presents our algorithms for the computation of quotients and factors of. Section 4 concludes.

2 Preliminaries

From now on we assume given a finite alphabet $\mathcal{A}$. We write $\mathcal{A}^*$ for the set of words of symbols from $\mathcal{A}$, and ϵ for the empty word. The *length* of a word w is denoted by $|w|$. A *language* is an arbitrary set of words, that is, a subset of $\mathcal{A}^*$. The *sum* of two languages is their union $E \cup F$. Their *product* $E.F$ is the set $\{vw \mid v \in E, w \in W\}$. The *asterate* E^* is the set $\bigcup_{n \in \mathbb{N}} E^n$ where $E^0 = \{\epsilon\}$ and $E^{n+1} := E.E^n$. The operations $\cup$, ., * are called *regular operations*. The set $\mathbf{Reg}(\mathcal{A})$ of *regular languages* is the set of languages obtained from the languages $\emptyset$, $\{\epsilon\}$, and $\{a\}$ (where $a \in \mathcal{A}$) by a finite number of applications of regular operations.

The *symmetric* of a language L is the language $L^{\mathtt{s}} := \{a_n \dots a_1 \mid a_1 \dots a_n \in L\}$. Regular languages are closed under intersection, complementation, and symmetry, that is, $L_1 \cap L_2, \mathcal{A}^* - L, L^{\mathtt{s}} \in \mathbf{Reg}(\mathcal{A})$ whenever $L, L_1, L_2 \in \mathbf{Reg}(\mathcal{A})$.

The set $\mathcal{T}(\mathcal{A})$ of *regular expressions* is defined recursively as follows:

- The symbols of $\mathcal{A}$, and 0 and 1 are regular expressions;
- If r and s are regular expressions, then so are $r \cdot s$, $r + s$ and $s^\star$.

The *alphabetic width* of $r \in \mathcal{T}(\mathcal{A})$, denoted by $\|r\|$, is the number of all occurrences of symbols from $\mathcal{A}$ in r. The *symmetric* of $r \in \mathcal{T}(\mathcal{A})$ is the regular expression $r^{\mathtt{s}}$ defined by: $r^{\mathtt{s}} := r$ if $r \in \{0,1\} \cup \mathcal{A}$; $(r+s)^{\mathtt{s}} := r^{\mathtt{s}} + s^{\mathtt{s}}$; $(r \cdot s)^{\mathtt{s}} := s^{\mathtt{s}} \cdot r^{\mathtt{s}}$; and $(r^\star)^{\mathtt{s}} := (r^{\mathtt{s}})^*$.

A regular expression r denotes a regular language $[\![r]\!]$ and this interpretation is determined by the following surjective homomorphism $[\![\_]\!]$ from the free algebra $(\mathcal{T}(\mathcal{A}), \{+, \cdot, ^*\})$ to $(\mathbf{Reg}(\mathcal{A}), \{\cup, ., ^*\})$:

$$[\![0]\!] := \emptyset,\ [\![1]\!] := \{\epsilon\},\ [\![r \cdot s]\!] = [\![r]\!].[\![s]\!],\ [\![r + s]\!] := [\![r]\!] \cup [\![s]\!], \text{ and } [\![r^\star]\!] := [\![r]\!]^*.$$

Note that the equality $[\![r^{\mathtt{s}}]\!] = [\![r]\!]^{\mathtt{s}}$ holds for all $r \in \mathcal{T}(\mathcal{A})$. Two regular expressions r and s are *equivalent*, and we write $r \doteq s$, if $[\![r]\!] = [\![s]\!]$.

The function $o : \mathcal{T}(\mathcal{A}) \to \{0,1\}$ which yields the *constant part* $o(r)$ of a regular expression r, is defined by

$$\begin{array}{ll} o(0) = o(a) := 0 \quad (a \in \mathcal{A}) & o(r+s) := \max\{o(r), o(s)\} \\ o(r \cdot s) := \min\{o(r), o(s)\} & o(1) = o(r^\star) := 1 \end{array}$$

It is easy to see that $o(r) = 1$ if $\epsilon \in [\![r]\!]$, and $o(r) = 0$ otherwise.

The quotients of two languages E and F are defined as follows [2]:

- the *left quotient* of E w.r.t. F is $F^{-1}E := \{w \mid \exists v.(v \in F \wedge vw \in E)\}$.
- the *right quotient* of E w.r.t. F is $EF^{-1} := \{w \mid \exists v.(v \in F \wedge wv \in E)\}$.

If F is a singleton language $\{w\}$ then we write simply $w^{-1}E$ and Ew^{-1} instead of the more cumbersome notations $\{w\}^{-1}E$ and $E\{w\}^{-1}$. We call these sets the *word derivative* and *word antiderivative* of E with respect to w.

It is well known that a regular language E has only finitely many left quotients $F^{-1}E$ (even for irregular language F), and that these are all regular languages [4]. The proof of this result can be easily adapted to conclude the same property for right quotients.

2.1 Systems of Characteristic Equations

A system of linear equations [6] is a system of language equations

$$\begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix} = \boldsymbol{A}. \begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix} \cup \boldsymbol{B} \tag{1}$$

where $\boldsymbol{A}$ is an $n \times n$ matrix of languages over $\mathcal{A}$, and $\boldsymbol{B}$ is an $n \times 1$ matrix of languages over $\mathcal{A}$. We will write $\boldsymbol{A}_{i,j}$ for the element of $\boldsymbol{A}$ at position (i,j), and $\boldsymbol{B}_i$ for the element at position $(i,1)$ of the $n \times 1$ matrix $\boldsymbol{B}$.

The matrix $\boldsymbol{A}$ is called ϵ-*free* if ϵ does not belong to any of the elements of $\boldsymbol{A}$. It is well known that if $\boldsymbol{A}$ is ϵ-free then (1) has the unique solution

$$\begin{pmatrix} X_1 \\ \vdots \\ X_n \end{pmatrix} = \boldsymbol{A}^*.\boldsymbol{B},$$

where $\boldsymbol{A}^*$ is the *asterate* of matrix $\boldsymbol{A}$ in the algebra $\mathcal{M}_{n,n}(\mathbf{Reg}(\mathcal{A}))$ [4, 6]. In this case, we call (1) a *system of characteristic equations* of the language $(\boldsymbol{A}^*.\boldsymbol{B})_1$.

Antimirov's improvement of Brzozowski algorithm [1, Sect. 4.2] computes, for every $r \in \mathcal{T}(\mathcal{A})$, the following:

- A finite set $\partial_{\mathcal{A}^\star}(r) := \{r_1, \ldots, r_n\} \subseteq \mathcal{T}(\mathcal{A})$, called the partial word derivatives of r,
- An $n \times n$ linear matrix[1] $\boldsymbol{A}$ and an $n \times 1$ constant vector[2] $\boldsymbol{B}$

such that $n \leq \|r\| + 1$ and

$$\begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \doteq \boldsymbol{A} \cdot \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} + \boldsymbol{B}. \tag{2}$$

We recall from [1] that $\partial : \mathcal{A}^* \times \mathcal{T}(\mathcal{A}) \to \mathcal{P}(\mathcal{T}(\mathcal{A}))$ satisfies the conditions:

- $\partial_1(s) := s$, $\partial_{aw}(s) := \bigcup_{s' \in \partial_a(s)} \partial_w(s')$, $\partial_W(s) := \bigcup_{w \in W} \partial_w(s)$, and
- $w^{-1}[\![s]\!] = \bigcup_{s' \in \partial_w(s)} [\![s']\!]$

for all $a \in \mathcal{A}$, $w \in \mathcal{A}^*$, $W \subseteq \mathcal{A}^*$, and $s \in \mathcal{T}(\mathcal{A})$.

Note that the elements of $\boldsymbol{A}$ in (2) denote ϵ-free languages, therefore (2) denotes a system of characteristic equations for $[\![r_1]\!]$ with the unique solution

$$\begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \doteq \boldsymbol{A}^\star \cdot \boldsymbol{B}$$

where $\boldsymbol{A}^\star$ is the asterate of matrix $\boldsymbol{A}$ in the algebra $\mathcal{M}_{n,n}(\mathcal{T}(\mathcal{A}))$ [6].

Antimirov's algorithm identifies regular expressions modulo fine similarity, which is an equivalence relation weaker than "$\doteq$." Therefore, (2) may not be minimal in the number of equations among the systems of characteristic equations for $[\![r_1]\!]$. We can minimize the system if we identify regular expressions $r_1, \ldots, r_n$ modulo equivalence.

[1] A linear matrix is matrix whose elements are 0 or sums of elements from $\mathcal{A}$.
[2] A constant matrix has all entries 0 or 1.

2.2 Factors of Regular Languages

A product of languages $F_1.\ldots.F_n$ is a *subfactorization* of a language E if and only if $F_1.\ldots.F_n \subseteq E$. The languages $F_1, \ldots, F_n$ are called the *terms* of the subfactorization. A term F_i is *maximal* if it can not be increased without violating the language inclusion. A *factorization* of E is a subfactorization in which every term is maximal. A *factor* of E is any language which is a term in some factorization of E. A *left* (resp. *right*) factor of E is one which can be the leftmost (resp. rightmost) term in some factorization of E. The *left* (resp. *right*) factor of E with respect to F is the maximal term G in the subfactorization $G.F$ (resp. $F.G$) of E. We denote the left (resp. right) factor of E with respect to F by $E \lhd F$ (resp. $F \rhd E$).

We write $r \leq s$ for $r, s \in \mathcal{T}(\mathcal{A})$ iff $[\![r]\!] \subseteq [\![s]\!]$.

Example 1. Let $\mathcal{A} = \{a, b\}$ and $E = [\![a^\star b^\star]\!]$. There are four 2-term factorizations $F_1.F_2$ of E: $F_1 = [\![r]\!]$ and $F_2 = [\![s]\!]$ where $r, s \in \mathcal{T}(\mathcal{A})$ such that

$$\langle r, s\rangle \in \{\langle 0, (a+b)^\star\rangle, \langle a^\star, a^\star b^\star\rangle, \langle a^\star b^\star, b^\star\rangle, \langle (a+b)^\star, 0\rangle\}.$$

Note that $[\![0]\!].[\![(a+b)^\star]\!]$ is a factorization of E, but $[\![0]\!].[\![(a+b)^\star]\!] = \emptyset \neq E$. □

In general $r_1 \cdot r_2 \doteq r$ is neither implied by, nor implies, that $[\![r_1]\!].[\![r_2]\!]$ is a factorization of $[\![r]\!]$. For example, $a^\star \doteq r_1 \cdot r_2$ where $r_1 = 1 + a$ and $r_2 = a^\star$, but $[\![r_1]\!].[\![r_2]\!]$ is not a factorization of $a^\star$ because the term $[\![r_1]\!]$ is not maximal with respect to $\subseteq$.

We recall some basic results about language factors:

1. Any left factor is a left term in some 2-term factorization. Any factor is the central term in some 3-term factorization. Any right factor is the right term in some 2-term factorization. [4, Ch.6, Thm. 2]
2. The condition that $L.R$ be a factorization of E defines a 1-1 correspondence between the left and right factors. [4, Ch.6, Thm. 3]
3. A regular language has finitely many factors, which are regular languages. [4, Ch.6, Thm. 5]

Let $E \in \mathbf{Reg}(\mathcal{A})$ be a regular language such that $\{L_i.R_i \mid 1 \leq i \leq n\}$ are all 2-term factorizations of E. Obviously, every factor of E belongs to the set $\{E_{i,j} \mid 1 \leq i, j \leq n\}$ where $E_{i,j}$ is defined by the condition that $L_i.E_{i,j}.R_j$ be a subfactorization of E in which $E_{i,j}$ is maximal. The $n \times n$ matrix

$$\begin{pmatrix} E_{1,1} & \ldots & E_{1,n} \\ \vdots & \ddots & \vdots \\ E_{n,1} & \ldots & E_{n,n} \end{pmatrix}$$

is called the *factor matrix* of E, and has some remarkable properties [4]:

1. There exist unique indices $l, r \in \{1, \ldots, n\}$ such that $E = L_r = R_l = E_{l,r}$ and $L_i = E_{l,i}$, $R_i = E_{i,r}$ for all $i \in \{1, \ldots, n\}$.
2. $\epsilon \in E_{i,i}$ for all $i \in \{1, \ldots, n\}$.
3. $F_1.F_2 \subseteq E_{i,k}$ if and only if $F_1 \subseteq E_{i,j}$ and $F_2 \subseteq E_{j,k}$ for some $j \in \{1, \ldots, n\}$.
4. $F_1.\ldots.F_p \subseteq E$ if and only if there are indices $i_1, \ldots, i_p \in \{1, \ldots, n\}$ such that $F_1 \subseteq E_{l,i_1}$, $F_p \subseteq E_{i_{p-1},r}$, and $F_j \subseteq E_{i_{j-1},i_j}$ for all $1 < j < p$.

3 Computational Methods for Quotients and Factors

In this paper we are interested in symbolic algorithms that work directly on some regular expressions r and s for the languages E and F, and produce:

1. A regular expression for the left and right quotient of E with respect to F.
2. A set of regular expressions for the right factors of E, and a set of regular expressions for the left factors of E.
3. A regular expression for the left and right factor of E with respect to F.
4. Regular expressions for the factors of the factor matrix of E.

The remainder of this section describes our algorithms for these computations.

3.1 Regular Expressions for Left and Right Quotients

First, we investigate the computation of a regular expression rs^{-1} for $[\![r]\!][\![s]\!]^{-1}$. Note that, if $r, r_1, r_2, s \in \mathcal{T}(\mathcal{A})$ then $r \doteq o(r) + \sum_{a\in\mathcal{A}} \sum_{q\in\partial_a(r)} q$ and:

1. $rs^{-1} \doteq o(rs^{-1}) + \sum_{a\in\mathcal{A}} \sum_{q\in\partial_a(r)} a \cdot (qs^{-1})$.
2. $(r_1 + r_2)s^{-1} \doteq r_1 s^{-1} + r_2 s^{-1}$.

An immediate consequence of this fact is that, if

$$\begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \doteq \boldsymbol{A} \cdot \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} + \begin{pmatrix} o(r_1) \\ \vdots \\ o(r_n) \end{pmatrix}$$

is a system of characteristic equations for $[\![r_1]\!]$, then

$$\begin{pmatrix} r_1 s^{-1} \\ \vdots \\ r_n s^{-1} \end{pmatrix} \doteq \boldsymbol{A} \cdot \begin{pmatrix} r_1 s^{-1} \\ \vdots \\ r_n s^{-1} \end{pmatrix} + \begin{pmatrix} o(r_1 s^{-1}) \\ \vdots \\ o(r_n s^{-1}) \end{pmatrix}$$

is a system of characteristic equations for $[\![r_1 s^{-1}]\!]$. It follows that, if we manage to compute the vector

$$\boldsymbol{C} := \begin{pmatrix} o(r_1 s^{-1}) \\ \vdots \\ o(r_n s^{-1}) \end{pmatrix} \in \mathcal{M}_{n,1}(\{0,1\})$$

then we can define

$$\begin{pmatrix} r_1 s^{-1} \\ \vdots \\ r_n s^{-1} \end{pmatrix} := \boldsymbol{A}^\star \cdot \boldsymbol{C} \in \mathcal{M}_{n,1}(\mathcal{T}(\mathcal{A})).$$

In particular, we define $r_1 s^{-1}$ as $(\boldsymbol{A}^\star \cdot \boldsymbol{C})_1$.

At this stage, the only item that we still have to compute is vector $\boldsymbol{C}$. For this purpose, we consider the systems of characteristic equations

$$\begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \doteq \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} + \boldsymbol{A} \cdot \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix}$$
$$\begin{pmatrix} s_1 \\ \vdots \\ s_m \end{pmatrix} \doteq \begin{pmatrix} c_1 \\ \vdots \\ c_m \end{pmatrix} + \boldsymbol{B} \cdot \begin{pmatrix} s_1 \\ \vdots \\ s_m \end{pmatrix}$$

for $[\![r_1]\!]$ and $[\![s_1]\!]$ where $r_1 = r$ and $s_1 = s$. We define the monotone operator

$$\mathtt{F} : \mathcal{P}(\{r_1, \ldots, r_n\} \times \{s_1, \ldots, s_m\}) \to \mathcal{P}(\{r_1, \ldots, r_n\} \times \{s_1, \ldots, s_m\})$$

represented compactly by the following collection of inference rules

$$\frac{[b_i = 1 \wedge c_j = 1]}{\langle r_i, s_j \rangle} \quad \frac{\langle r_k, s_l \rangle \quad [a \in \mathcal{A} \wedge r_k \in \partial_a(r_i) \wedge s_l \in \partial_a(s_j)]}{\langle r_i, s_j \rangle}$$

Each rule states that if the pair above the bar is in the input set and the conditions within square brackets hold, then the pair below is in the output set.

$\mathtt{F}$ is a monotone operator defined on the finite cpo $(\mathcal{P}(\partial_{\mathcal{A}^*}(r) \times \partial_{\mathcal{A}^*}(s)), \subseteq)$. Therefore, the least fixed point $\mu\mathtt{F}$ of $\mathtt{F}$ is a finite computable set.

Lemma 1. *The least fixed point $\mu\mathtt{F}$ of $\mathtt{F}$ is the set $\{\langle r_i, s_j \rangle \mid o(r_i s_j^{-1}) = 1\}$.*

Proof. Let $M = \{\langle r_i, s_j \rangle \mid o(r_i s_j^{-1}) = 1\}$. We prove $\mu\mathtt{F} \subseteq M$ by induction of the length of the inference derivation.

If $\langle r_i, s_j \rangle \in \mu\mathtt{F}$ was deduced by the inference rule

$$\frac{[b_i = 1 \wedge c_j = 1]}{\langle r_i, s_j \rangle}$$

then $o(r_i) = b_i = 1 = c_j = o(s_j)$, therefore $\epsilon \in [\![r_i]\!] \cap [\![s_j]\!]$ and thus $\epsilon \in [\![r_i]\!][\![s_j]\!]^{-1} = [\![r_i s_j^{-1}]\!]$. Hence $o(r_i s_j^{-1}) = 1$, therefore $\langle r_i, s_j \rangle \in M$.

If $\langle r_i, s_j \rangle \in \mu\mathtt{F}$ was deduced by a derivation with the last inference rule

$$\frac{\langle r_k, s_l \rangle \quad [a \in \mathcal{A} \wedge r_k \in \partial_a(r_i) \wedge s_l \in \partial_a(s_j)]}{\langle r_i, s_j \rangle}$$

then $\langle r_k, s_l \rangle \in \mu\mathtt{F}$, $a \cdot r_k \leq r_i$ and $a \cdot s_l \leq s_j$. By induction hypothesis we have $o(r_k s_l^{-1}) = 1$. But $o(r_k s_l^{-1}) = 1$ iff $[\![r_k]\!] \cap [\![s_l]\!] \neq \emptyset$ iff $[\![a \cdot r_k]\!] \cap [\![a \cdot s_l]\!] \neq \emptyset$. Since $[\![a \cdot r_k]\!] \cap [\![a \cdot s_l]\!] \subseteq [\![r_i]\!] \cap [\![s_j]\!]$, we conclude $\epsilon \in [\![r_i]\!][\![s_j]\!]^{-1}$, hence $o(r_i s_j^{-1}) = 1$.

Next, we prove $M \subseteq \mu\mathtt{F}$. Note that $\langle s_i, r_j \rangle \in M$ iff there exists a word $w \in [\![r_i]\!] \cap [\![s_j]\!]$, and thus we can define the complexity measure $|\langle s_i, r_j \rangle|$ of every $\langle r_i, s_j \rangle \in M$ as the minimal length of a word of $[\![r_i]\!] \cap [\![s_j]\!]$. We prove $M \subseteq \mu\mathtt{F}$ by induction on the complexity measure of elements of M.

If $\langle r_i, s_j\rangle \in M$ and $|\langle r_i, s_j\rangle| = 0$ then $\epsilon \in [\![r_i]\!] \cap [\![s_j]\!]$. Thus $b_i = o(r_i) = 1$, $c_j = o(s_j) = 1$, and we can perform the derivation

$$\frac{[b_i = 1 \wedge c_j = 1]}{\langle r_i, s_j\rangle}$$

to deduce that $\langle r_i, s_j\rangle \in \mu\mathtt{F}$. If $\langle r_i, s_j\rangle \in M$ and $|\langle r_i, s_j\rangle| = p > 0$ then $\epsilon \notin [\![r_i]\!] \cap [\![s_j]\!]$ and there exists $w = aw_1 \in [\![r_i]\!] \cap [\![s_j]\!]$ of length p with $a \in \mathcal{A}$. This implies $w_1 \in [\![r_k]\!]$ for some $r_k \in \partial_a(r_i)$ and $w_1 \in s_l$ for some $s_l \in \partial_a(s_j)$. Thus $w_1 \in [\![r_k]\!] \cap [\![s_l]\!]$, and we learn that $\langle r_k, s_l\rangle \in M$. Since w_1 has length $p-1$, we learn that $|\langle r_k, s_l\rangle| \leq p - 1 < p$ and we can apply the induction hypothesis to conclude that $\langle r_k, s_l\rangle \in \mu\mathtt{F}$. Finally, we can use the facts that $r_k \in \partial_a(r_i)$ and $s_l \in \partial_a(s_l)$ and apply the inference step $\dfrac{\langle r_k, s_l\rangle}{\langle r_i, s_j\rangle}$ to conclude $\langle r_i, s_j\rangle \in \mu\mathtt{F}$. □

We conclude this section by noting that our algorithm for the computation of regular expressions for right quotients can be turned easily into an algorithm for the computation of regular expressions for left quotients: It is sufficient to observe that we can define $s^{-1}r := (r^{\mathtt{s}}(s^{\mathtt{s}})^{-1})^{\mathtt{s}}$.

Example 2. Let $r, s \in \mathcal{T}(\{a, b\})$, $r = (a + b)^\star \cdot a$ and $s = b \cdot a^\star$. In order to compute rs^{-1}, we compute systems of characteristic equations for $[\![r]\!]$ and $[\![s]\!]$:

$$\begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \doteq \begin{pmatrix} a+b & a \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{ where } \begin{cases} r_1 = r \\ r_2 = 1 \end{cases},$$
$$\begin{pmatrix} s_1 \\ s_2 \end{pmatrix} \doteq \begin{pmatrix} 0 & b \\ 0 & a \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ s_2 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{ where } \begin{cases} s_1 = s \\ s_2 = a^\star \end{cases},$$

and conclude that

$$\begin{pmatrix} r_1 s_1^{-1} \\ r_2 s_1^{-1} \end{pmatrix} \doteq \begin{pmatrix} b & a \\ b & a \end{pmatrix} \cdot \begin{pmatrix} r_1 s_1^{-1} \\ r_2 s_1^{-1} \end{pmatrix} + \begin{pmatrix} o(r_1 s_1^{-1}) \\ o(r_2 s^{-1}) \end{pmatrix}.$$

In this case $\mathtt{F} : \mathcal{P}(\{r_1, r_2\} \times \{s_1, s_2\}) \to \mathcal{P}(\{r_1, r_2\} \times \{s_1, s_2\})$ is the monotone operator defined by the inference rules

$$\frac{\langle r_1, s_2\rangle}{\langle r_1, s_1\rangle} \quad \frac{\langle r_1, s_2\rangle}{\langle r_1, s_2\rangle} \quad \frac{\langle r_2, s_2\rangle}{\langle r_1, s_2\rangle} \quad \frac{}{\langle r_2, s_2\rangle}$$

and $\mu\mathtt{F} = \{\langle r_2, s_2\rangle, \langle r_1, s_2\rangle, \langle r_1, s_1\rangle\}$. By Lemma 1 we obtain $o(r_1 s^{-1}) = 1$, $o(r_2 s^{-1}) = 0$. Since

$$A^\star = \begin{pmatrix} (a+b)^\star & (a+b)^\star \cdot a \\ 0 & 1 \end{pmatrix}$$

and $r_1 s_1^{-1}$ is the first component of the column vector $A^\star \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, we conclude that $rs^{-1} = r_1 s_1^{-1} = (a + b)^\star$. □

3.2 Regular Expressions for the Sets of Left and Right Factors

We are looking for an algorithm that computes a finite set $RF(r)$ of regular expressions such that every right factor of $[\![r]\!]$ is represented by some element of $RF(r)$. Let $E = [\![r]\!]$, $\partial_{\mathcal{A}^*}(r) = \{r_1, \ldots, r_n\}$ with $r_1 = r$, and

$$\begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \doteq \boldsymbol{A} \cdot \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} + \boldsymbol{C} \tag{3}$$

be a system of characteristic equations for E. We denote by $D_{\mathcal{A}^*}(r)$ the computable set $\{\partial_w(r) \mid w \in \mathcal{A}^*\}$ and note that

- $D_{\mathcal{A}^*}(r)$ has at most $2^{|\partial_{\mathcal{A}^*}(r)|} \leq 2^{\|r\|+1}$ elements,
- $w^{-1}[\![r]\!] = \bigcup_{s \in \partial_w(r)} [\![s]\!]$ for every $w \in \mathcal{A}^*$, and
- $\{w^{-1}[\![r]\!] \mid w \in \mathcal{A}^*\} = \{\bigcup_{s \in M} [\![s]\!] \mid M \in D_{\mathcal{A}^*}(r)\}$.

F is a right factor of $[\![r]\!]$ if and only if there exists a subfactorization $L.F$ of $[\![r]\!]$ where F is maximal; that is to say, if and only if $F = \bigcap_{w \in L} w^{-1}[\![r]\!]$. This shows that the set of right factors of $[\![r]\!]$ is

$$\left\{ \bigcap_{M \in S} \left(\bigcup_{s \in M} [\![s]\!] \right) \,\middle|\, S \subseteq D_{\mathcal{A}^*}(r) \right\}.$$

Since $M \subseteq \partial_{\mathcal{A}^*}(r)$ for every $M \in D_{\mathcal{A}^*}(r)$, we learn that every right factor can be expressed as a finite union of intersections of elements of $\partial_{\mathcal{A}^*}(r)$. More formally, we can express every right factor of $[\![r]\!]$ as

$$\bigcup_{M \in \mathfrak{M}} \bigcap_{G \in M} [\![G]\!] = \bigcup_{M \in \mathfrak{M}_0} \bigcap_{G \in M} [\![G]\!]$$

where $\mathfrak{M} \subseteq \mathcal{P}(\partial_{\mathcal{A}^*}(r))$ and $\mathfrak{M}_0$ is the set of $\subseteq$-minimal elements of $\mathfrak{M}$.

Thus, in order to compute regular expressions for the right factors of r it is sufficient to be able to compute regular expressions for the languages $\bigcap_{s \in K} [\![s]\!]$ when $K \subseteq \partial_{\mathcal{A}^*}(r)$. We can generate a system of characteristic equations for $\bigcap_{s \in K} [\![s]\!]$ as follows. Suppose $\mathcal{A} = \{a_1, \ldots, a_m\}$ and $K = \{r_{k_1}, \ldots, r_{k_p}\} \subseteq \{r_1, \ldots, r_n\}$. Then

$$r_{k_l} \doteq \boldsymbol{C}_{k_l} + \sum_{j=1}^{n} \boldsymbol{A}_{k_l, j} \cdot r_j \qquad (1 \leq l \leq p)$$

and by intersecting these p equations we obtain

$$\bigcap_{s \in K} s \doteq \min\{\boldsymbol{C}_{k_l} \mid 1 \leq l \leq p\} + \sum_{j_1=1}^{n} \ldots \sum_{j_p=1}^{n} \boldsymbol{M}^{k_1,\ldots,k_p}_{j_1,\ldots,j_p} \cdot (r_{j_1} \cap \ldots \cap r_{j_p}) \tag{4}$$

where

$$\boldsymbol{M}^{k_1,\ldots,k_p}_{j_1,\ldots,j_p} := \begin{cases} a \text{ if } \boldsymbol{A}_{k_l,j_l} = a \in \mathcal{A} \text{ for all } l \in \{1,\ldots,p\}, \\ 0 \text{ otherwise.} \end{cases}$$

We can produce such an equation for every extended regular expression $\bigcap_{s\in K} s$. The system af all these equations, which has (4) as first equation is obviously a system of characteristic equations for $\bigcap_{s\in K}[\![s]\!]$. By solving it, we get a regular expression for the right factor $\bigcap_{s\in K}[\![s]\!]$. Before solving the system of characteristic equations for $\bigcap_{s\in K}[\![s]\!]$, it is desirable to reduce its size at much as possible. The following criteria can be used for this purpose:

- Identify extended regular expressions of the form $\bigcap_{s\in K'} s$ modulo associativity, commutativity, and idempotence of intersection.
- Identify with 0 the expressions $\bigcap_{s\in K'} s$ with $0 \in K'$.
- Keep only the minimal set of equations necessary for a system of characteristic equations of E.

The algorithm described in the previous section can be used to compute regular expressions for the left factors of E too: It is easy to verify that the set $LF(r) := \{s^{\mathtt{s}} \mid s \in RF(r^{\mathtt{s}})\}$ consists of regular expressions for all left factors of $[\![r]\!]$.

3.3 Regular Expressions for Left and Right Factors between Regular Languages

Here we address the following problem:

Given: $r_1, s_1 \in \mathcal{T}(\mathcal{A})$,
Compute: $r' \in \mathcal{T}(\mathcal{A})$ such that $[\![r']\!] = ([\![s_1]\!] \rhd [\![r_1]\!])$.

It is not hard to see that

$$([\![s_1]\!] \rhd [\![r_1]\!]) = \bigcap_{w\in[\![s_1]\!]} w^{-1}[\![r_1]\!] = \bigcap_{n\in\mathbb{N}} \bigcap_{\substack{w\in[\![s_1]\!] \\ |w|=n}} w^{-1}[\![r_1]\!].$$

Let's consider the ternary relation $s \rhd M \rightsquigarrow N$ with $s \in \partial_{\mathcal{A}^*}(s_1)$, $M, N \in \mathcal{P}(\partial_{\mathcal{A}^*}(r_1))$, defined inductively by

$$\frac{o(s) = 1}{s \rhd M \rightsquigarrow M} \qquad \frac{s' \rhd M' \rightsquigarrow N \quad [s' \in \partial_a(s) \wedge M' = \bigcup_{r\in M} \partial_a(r)]}{s \rhd M \rightsquigarrow N}.$$

Intuitively, the relation $s \rhd M \rightsquigarrow N$ holds iff there exists $w \in [\![s]\!]$ such that $N = \bigcup_{r\in M} \partial_w(r)$. This ternary relation is decidable because it is defined inductively on finite sets.

Lemma 2. *If $S \in \partial_{\mathcal{A}^*}(s_1)$ and $M \subseteq \partial_{\mathcal{A}^*}(r_1)$, then*

$$\left([\![s]\!] \rhd \bigcup_{r\in M} [\![r]\!]\right) = \bigcap_{s\rhd M\rightsquigarrow N} \left(\bigcup_{r\in N} [\![r]\!]\right).$$

Proof. For every $n \in \mathbb{N}$ we define the ternary relation

$$s \rhd M \rightsquigarrow_n N :\Leftrightarrow \text{there is a derivation of length } n \text{ of } s \rhd M \rightsquigarrow N$$

and note that

$$\left([\![s]\!] \rhd \bigcup_{r \in M} [\![r]\!] \right) = \bigcap_{n \in \mathbb{N}} \bigcap_{\substack{w \in [\![s]\!] \\ |w| = n}} w^{-1} \left(\bigcup_{r \in M} [\![r]\!] \right),$$

$$\bigcap_{s \rhd M \rightsquigarrow N} \left(\bigcup_{r \in N} [\![r]\!] \right) = \bigcap_{n \in \mathbb{N}} \bigcap_{s \rhd M \rightsquigarrow_n N} \left(\bigcup_{r \in N} [\![r]\!] \right).$$

Thus it is sufficient to prove that

$$\left\{ w^{-1} \left(\bigcup_{r \in M} [\![r]\!] \right) \,\middle|\, w \in [\![s]\!] \wedge |w| = n \right\} = \left\{ \bigcup_{r \in N} [\![r]\!] \,\middle|\, s \rhd M \rightsquigarrow_n N \right\}$$

holds for every $s \in \partial_{\mathcal{A}^*}(s_1)$, $M \subseteq \partial_{\mathcal{A}^*}(r_1)$, and $n \in \mathbb{N}$. This fact can be proved easily by induction on n. □

We have ended up with the following algorithm for the computation of a regular expression for $[\![s_1]\!] \rhd [\![r_1]\!]$:

1. Compute the finite set $\mathfrak{N} := \{N \mid s_1 \rhd \{r_1\} \rightsquigarrow N\}$.
2. Compute r' as regular expression for the regular language $\bigcap_{N \in \mathfrak{N}} \bigcup_{r \in N} [\![r]\!]$. Since $\mathfrak{N}$ consists of subsets of $\partial_{\mathcal{A}^*}(r_1)$, we can use the intersection elimination algorithm presented in Sect. 3.2 to compute r'. We will denote the regular expression r' computed in this way by $s_1 \rhd r_1$, and call it the *product derivative* of r_1 w.r.t. s_1.

This algorithm can be turned easily into an algorithm for the computation of a regular expression for $[\![r]\!] \lhd [\![s]\!]$. More exactly, we observe that X is maximal in a subfactorization $X.[\![s]\!] \subseteq [\![r]\!]$ iff $X^{\mathtt{s}}$ is maximal in a subfactorization $[\![s^{\mathtt{s}}]\!].X^{\mathtt{s}} \subseteq r^{\mathtt{s}}$. Thus, we can use the previous algorithm with inputs $r^{\mathtt{s}}$ and $s^{\mathtt{s}}$ to compute a regular expression r' for $[\![r^{\mathtt{s}}]\!] \rhd [\![s^{\mathtt{s}}]\!]$, and conclude that $(r')^{\mathtt{s}}$ is a regular expression for $[\![r]\!] \lhd [\![s]\!]$.

3.4 Regular Expressions for Factors

This computation of regular expressions for the factor matrix of $[\![r]\!]$ can be carried out with the algorithms described so far as follows:

1. First, compute $LF(r)$ using the algorithm from Sect. 3.2. Suppose $LF(r) = \{l_1, \ldots, l_n\}$.
2. Next, compute $RF(r) := \{l_1 \rhd r, \ldots, l_n \rhd r\}$ using the algorithm described in Sect. 3.3. Let $r_i := (l_i \rhd r)$ for $1 \leq i \leq n$.
3. The $n \times n$ factor matrix $(E_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}}$ has $E_{i,j}$ maximal in the 3-term subfactorization $[\![l_i]\!].E_{i,j}.[\![r_j]\!] \subseteq [\![r]\!]$ for every $1 \leq i, j \leq n$. It follows that $E_{i,j} = [\![r'_{i,j}]\!]$ with $r'_{i,j} := l_i \rhd l_j$ for every $i, j \in \{1, \ldots, n\}$.

4 Conclusion

We have shown how the computation of quotients, factors, and product derivatives of regular languages represented by regular expressions can be reduced to computation and solving of systems of characteristic equations. In this way we produce regular expressions for all these operations.

An algorithm for the computation of product derivative of regular languages was proposed recently in [7]. It computes the greatest fixed point of a continuous operator which renders the result as a finite intersection of languages represented by regular expressions. Our computational method differs from [7] in two important respects: (1) it relies on the computation of the least fixed point of a monotone operator on a finite lattice to produce a representation of the product derivative as union of intersections of partial derivatives of a regular expression; and (2) proposes an algorithm to compute regular expressions for the intersection of partial derivatives via computations and solving of systems of characteristic equations.

References

1. V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155:291–319, 1996.
2. J. Berstel. *Transductions and Context-Free Languages.* B.G. Teubner Stuttgart, 1979.
3. J. A. Brzozowski. Derivatives of regular expressions. *Journal of the Association for Computing Machinery*, 11(4):481–494, October 1964.
4. J. H. Conway. *Regular Algebra and Finite Machines.* Mathematics series. Chapman and Hall, 1971.
5. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Series in Computer Science. Addison-Wesley Publishing Company, Inc., 1979.
6. D. C. Kozen. *Automata and Computability.* Undergraduate Texts in Computer Science. Springer-Verlag New York, Inc., 1997.
7. T. Suzuki and S. Okui. Product derivatives of regular expressions. *ISPJ Online Transactions*, 1:53–65, July 2008.

Copy-on-Write in the PHP Language

Akihiko Tozawa, Michiaki Tatsubori and Tamiya Onodera[1]

IBM Research, Tokyo Research Laboratory

Yasuhiko Minamide[2]

Department of Computer Science, University of Tsukuba

Abstract

PHP is a popular language for server-side applications. In PHP, assignment to variables copies the assigned values, according to its so-called *copy-on-assignment* semantics. In contrast, a typical PHP implementation uses a *copy-on-write* scheme to reduce the copy overhead by delaying copies as much as possible. This leads us to ask if the semantics and implementation of PHP coincide, and actually this is not the case in the presence of sharings within values. In this paper, we describe the copy-on-assignment semantics with three possible strategies to copy values containing sharings. The current PHP implementation has inconsistencies with these semantics, caused by its naïve use of copy-on-write. We fix this problem by the novel *mostly copy-on-write* scheme, making the copy-on-write implementations faithful to the semantics. We prove that our copy-on-write implementations are correct, using bisimulation with the copy-on-assignment semantics. (The longer version of this paper is published at Principles of Programming Languages (POPL) 2009.)

Keywords: Programming Languages, PHP, Semantics, Graph Rewriting

1 Introduction

Copy-on-write is a classic optimization technique, which delays the copy of data until the write to it. One example of copy-on-write is found in the UNIX *fork*, where the process-local memory is the local data that should be copied from the address space of the original process to the space of the new process by the fork operation. In modern UNIX systems, this copy is usually delayed by copy-on-write.

Another example is found in the PHP language, a popular scripting language for server-side Web applications. Here is an example with PHP's associative arrays.

```
$r["box"] = "gizmo";
$l = $r;                    // assignment from $r to $l
$l["box"] = "gremlin";
echo $r["box"];             // prints out gizmo
```

[1] atozawa@jp.ibm.com, mich@acm.org and tonodera@jp.ibm.com

[2] minamide@cs.tsukuba.ac.jp

©2009 Published by Elsevier Science B. V.

The write to $l at Line 3, following the assignment $l = $r, only has local effects on $l which cannot be seen from $r. The behavior in PHP is called *copy-on-assignment*, since the value of $r seems to be copied before it is passed to $l. We can consider the copy-on-write technique to implement this behavior. Indeed, the by far dominant PHP runtime, called the Zend runtime [3], employs copy-on-write and delays the above copy until the write at Line 3.

Now, our question is as follows. The copy-on-write is considered as a runtime optimization technique reducing useless copies. Then, does the use of copy-on-write preserve the equivalent behavior to the original, copy-on-assignment semantics? In fact, it is not the case in the presence of the mechanism of reference assignment =&, which declares sharing between two locations.

```
$r["box"] = "gizmo";
$x =& $r["box"];          // creates a shairng inside $r
$l = $r;                  // copies $r
$l["box"] = "gremlin";
echo $r["box"];           // what should it be ?
```

The result of this program should reflect how exactly PHP copies arrays when they contain sharing. Our discussion will start from clarifying such PHP's copy semantics.

In this paper, we investigate the semantics and implementation of PHP focusing on the copy-on-write technique and its problems. Our contributions in this paper are as follows.

- We develop three *copy-on-assignment* operational semantics of PHP, each differing in their copy strategies, i.e., how sharings inside arrays are copied. Three copy strategies are called *shallow copy*, *graphical copy*, and *deep copy*. To capture sharings inside values, our formal model uses *graphs* and their rewriting [BS92].
- We identify several problems in the current PHP implementation, including the inconsistency from the copy-on-assignment semantics. In particular, we point out the *inversion of execution order* problem, which is caused by the copy-on-write optimization.
- We propose copy-on-write implementations of PHP based on the novel *mostly copy-on-write* scheme, which fixes the inversion problem by adding moderate overhead to the implementation. This fix works for all three copy strategies. We prove that the corresponding copy-on-assignment and mostly copy-on-write models coincide using a *bisimulation* proof.

References

[BS92] Erik Barendsen and Sjaak Smetsers. Graph rewriting and copying. Technical Report 92-20, University of Nijmegen, 1992.

[3] Available at http://www.php.net.

Finding Bugs in AspectJ is not Difficult

Haihao Shen, Sai Zhang, Jianjun Zhao

School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{haihaoshen,saizhang,zhao-jj}@sjtu.edu.cn

1 Introduction

Static analysis for software defect detection is a promising technique to improve software quality. Because of the sheer complexity of modern programming languages, the potential for misuse of language features, API rules or simply bad programming practice may be enormous. Static analysis techniques can explore abstractions of all possible program behaviors, and thus are not limited by the quality of test cases in order to be effective. Static analysis tools, such as [11, 5, 8, 7, 12], serve an important role in raising the awareness of developers about subtle correctness issues. In addition to finding existing bugs, these tools can also help programmers to prevent future defects. FindBugs [5], one of the most popular static analysis tools, is becoming widely used in Java community. FindBugs implements a set of bug *detectors* for a variety of common *bug patterns* (code idioms that are likely to be errors [13]), and uses them to find a significant number of bugs in real-world applications and libraries [10, 9].

Aspect-Oriented Programming (AOP) [14] has been proposed as a technique for improving separation of concerns in software design and implementation. It is gaining popularity with the wider adoption of languages such as AspectJ. AspectJ is a seamless extension of Java. An AspectJ program can be divided into two parts: *base code* which includes classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modeling cross-cutting concerns in the program. However, though the state-of-the-art aspect-oriented programming environments (such as AJDT [3] in the eclipse [4] IDE) provide powerful capability to check the syntactic or grammar errors in AspectJ programs, they fail to detect potential semantic defects in software systems. For example, the type checker used in AJDT only checks the syntactic correctness of the program, but fails to identify or even to generate a warning about the *type conflicts* introduced by an aspect. In such cases, once the class containing an introduced field with a conflicting type is instantiated, the whole program will be terminated abruptly.

Although the executable code of an AspectJ program is pure Java bytecode, the existing bug patterns and defect detection tools for Java bytecode might not

* This work was presented at the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2008), as a regular paper with the title "XFindBugs: eXtended FindBugs for AspectJ" by Haihao Shen, Sai Zhang, Jianjun Zhao, Jianhong Fang, and Shiyuan Rao.

be applied directly. In addition to the bytecode that corresponds to the source code (e.g., to bodies of advices), the compiled bytecode of an AspectJ program contains extra code inserted by the compiler during the weaving process. After weaving, the source code level *aspect*, *advice* or *intertype declaration* information has been translated into pure Java bytecode instructions, and therefore is no longer preserved. In fact, in our experimental study, none of the bugs found by XFindBugs can be detected directly by FindBugs.

In this talk, we introduce XFindBugs, an eXtended FindBugs for AspectJ. XFindBugs defines a catalog of 17 bug patterns for aspect-oriented features, and implements a set of bug detectors on top of the FindBugs analysis framework. Bug patterns abstract common misunderstandings of language features, API rules and bad programming practice. They help programmers get a better understanding of how to write bug-free code. We also perform an empirical evaluation of XFindBugs on several AspectJ benchmarks and third-party large-scale applications (like GlassBox [6], AJHotDraw [1], and AJHSQLDB [2]). XFindBugs confirms 7 reported bugs and finds 257 previously unknown defects in these subjects, some of which may even result in a software crash. The experiment also indicates that the bug patterns XFindBugs supports exist in real-world software systems, even in mature AspectJ applications by experienced programmers.

In summary, the main contributions of this paper are: **(1)** a systematic catalog of bug patterns for AspectJ programs, **(2)** design and implementation of XFindBugs, a static defect detection tool for AspectJ software, and **(3)** an empirical evaluation of XFindBugs on over 300KLOC, which evidences the practical issues.

Acknowledgements. This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Cheng Zhang, Qingzhou Luo, and Xin Huang for their valuable discussions on this work.

References

1. AJHotDraw. http://sourceforge.net/projects/ajhotdraw.
2. AJHSQLDB. http://sourceforge.net/projects/ajhsqldb.
3. Aspectj development tools (ajdt). http://www.eclipse.org/ajdt/.
4. Eclipse. http://www.eclipse.org/.
5. FindBugs. http://findbugs.sourceforge.net/.
6. Glassbox. http://www.glassbox.com/.
7. JLint. http://jlint.sourceforge.net/.
8. PMD. http://pmd.sourceforge.net/.
9. N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.
10. N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou. Using FindBugs on production software. *OOPSLA 07: Companion to the 22nd ACM SIGPLAN*

conference on Object oriented programming systems and applications companion, 2007.
11. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.
12. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *In Tenth International Workshop on Model Checking of Software (SPIN), volume 2648 of Lecture Notes in Computer Science, pages 235–239. Springer-Verlag, 2003.*, 2003.
13. D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.
14. G. Kiczales et al. Aspect-oriented programming. *ACM SIGSOFT Software Engineering Notes*, 26(5):313, 2001.

Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the ENTCS Macro Home Page.

AOJS: Aspect-Oriented JavaScript Programming Framework [1]

Hironori Washizaki, Atsuto Kubo, Tomohiko Mizumachi,
Kazuki Eguchi, Yoshiaki Fukazawa[2]

Dept. Computer Science and Engineering, Waseda University
3-4-1, Okubo, Shinjuku-ku, Tokyo, 169-8555, Japan

Nobukazu Yoshioka, Hideyuki Kanuka, Toshihiro Kodaka,
Nobuhide Sugimoto, Yoichi Nagai, and Rieko Yamamoto

National Institute of Infomatics, Hitachi, Ltd., Fujitsu Laboratories ltd.,
Toshiba Solutions Corporation, NEC Corporation, and Fujitsu Laboratories ltd.

JavaScript (ECMAScript[1]) is a popular scripting language that is particularly useful for client-side programming together with HTML/XML on the Web. In JavaScript programming, there are many concerns (such as logging and Ajax-based functions) that cannot be encapsulated and separated into independent modules due to the limitation of JavaScript's modularization mechanism. For example, when conducting a beta-test of a typical web application with JavaScript program, it might be necessary to log all value changes of specific variables and send each log to remote sever at runtime because of variety of web client environments. Embedding remote-logging function codes into each location where variable substitutions will take place is the traditional way for realizing such logging function; however since the additional codes scatters and tangle with other concerns' codes, the maintainability of the program will decrease significantly.

To encapsulate and separate realizations of such crosscutting concerns into independent modules, there are several Aspect-Oriented Programming (AOP[2]) frameworks for JavaScript, such as Aspectjs[3] and Google Ajaxpect[4]. However, regarding all of conventional frameworks, it is necessary to modify the target program to include extended library and/or describe aspects in itself. Moreover none of conventional frameworks can specify the location where variable substitutions take place as joinpoints for weaving JavaScript codes.

[1] This paper has been accepted for demonstration at AOSD 2009. The extended content is under review for 8th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software.

[2] Contact to Hironori Washizaki, Email: washizaki@waseda.jp

©2009 Published by Elsevier Science B. V.

To solves these problems, we propose an Aspect-Oriented JavaScript programming framework, named "AOJS". AOJS is based on the proxy architecture for weaving aspects and the joinpoint model[5] for specifying program locations where the aspects will be weaved.

Firstly, AOJS realize the complete separation of aspects and target programs by the proxy-based runtime weaving. This design leads to the fact that AOJS always ensures the consistency between programs and the ones with aspects weaved by the proxy. Moreover the client does not have to consider AOP nor the weaving process when requesting web pages with JavaScript programs; it is only necessary to know the proxy's URL. Therefore it is easy to weave/remove aspects at runtime by only changing the URL for accessing. Figure 1(a) shows the architecture of AOJS. AOJS is realized as a server that communicates with web clients that request web pages via HTTP and web servers that store/provide original web pages and JavaScript programs. AOJS server consists of two parts: the reverse proxy for judging necessity of weaving and redirecting requests/outputs, and the weaver for weaving.

Secondly, AOJS allows programmers to specify any variable substitution, function execution and file initilization as a joinpoint by using corresponding pointcuts: `<var>`, `<function>` and `<initializeFile>` written in an aspect file in the form of XML. For the specified joinpoints, AOJS can weave both of before and after advices that will be performed before/after the target joinpoint's execution. Figure 1(b) shows the example of weaving when a variable substitution has been specified as a joinpoint. In the figure, the code portion surrounded by the dashed line will be replaced by the code for replacement based on the template. These replacements add new behavior into the target program while keeping the original functionality.

We conducted some experimental evaluations regarding the functionality and runtime performance of AOJS, and confirmed that AOJS has enough ability to specify joinpoints including variable substitutions. Moreover, we also confirmed the runtime performance can be improved to a practical level by adding a cash proxy in front of the reverse proxy.

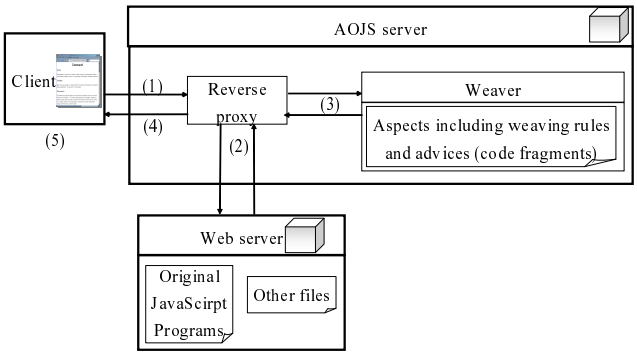
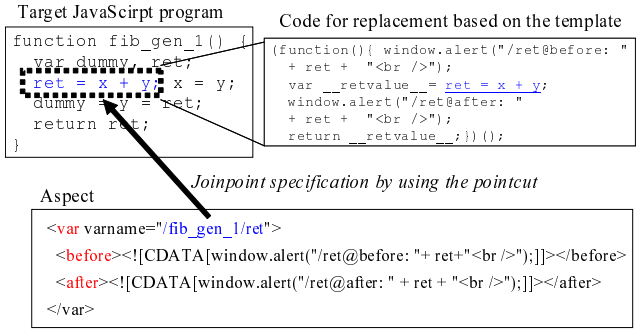


Fig. 1. (a) Architecture of AOJS



(b) Weaving mechanism for variable substitution

References

[1] ISO/IEC 16262:2002, Information technology - ECMAScript language specification, 2002.
[2] Gregor Kiczales, et al.: Aspect-oriented programming, Proc. European Conference on Object-Oriented Programming (ECOOP), pp.220–242, 1997.
[3] LECACHEUR Sebastien: Aspectjs, `http://zer0.free.fr/aspectjs/`
[4] Google: Ajaxpect: Aspect-Oriented Programming for Ajax, `http://code.google.com/p/ajaxpect/`
[5] Gregor Kiczales, et al.: An Overview of AspectJ, Proc. European Conference on Object-Oriented Programming (ECOOP), pp.327–353, 2001.

Formalization and Specification of Geometric Knowledge Objects

Dongming Wang[1,2]

Laboratoire d'Informatique de Paris 6, Université Pierre et Marie Curie – CNRS, 104 avenue du Président Kennedy, F-75016 Paris, France
LMIB – SKLSDE – School of Mathematics and Systems Science, Beihang University, Beijing 100191, China

Abstract

This note presents our work in progress on the identification, formalization, structuring, and specification of geometric knowledge objects for the purpose of electronic storage and management. We classify geometric knowledge into knowledge objects according to how knowledge has been accumulated and represented in the geometric literature, formalize geometric knowledge objects using the language of predicate logic with embedded knowledge, and organize them by modeling the hierarchic structure of relations among them. Some examples of formal specification for geometric knowledge objects are given to illustrate our approach. The underlying idea of this approach has been used successfully for automated geometric reasoning, knowledge base creation, and electronic document generation.

Keywords: Embedded knowledge, formal specification, knowledge management, predicate logic.

1 Introduction

Geometry has been the target of study of computer scientists since the late 1950s. The involvement of algebraic computation, logical reasoning, and graphical representation makes geometry a unique touchstone to test the suitability and power of modern computer software and hardware and of computational methods implemented therein for scientific problem solving. However, the capabilities of currently available software for the manipulation of symbolic geometric objects, quantities, and relations are very limited, in contrast to those of computer algebra systems such as Maple and Mathematica for the manipulation of algebraic objects. Algebraic objects such as numbers, polynomials, and transcendental functions are purely symbols that are endowed with meanings by using formal definitions. The problem of representation and formal manipulation of algebraic objects on computer has been thoroughly studied. It is now well understood what kinds of data structures and computational mechanisms should be implemented for algebraic manipulation.

In geometry we may understand an arbitrary triangle as a shape formed by connecting three arbitrary points with three line segments. Then natural questions arise: what is a shape, what is a point, and what is a segment? What does "connecting" mean? Is a point also a (degenerated) triangle or a circle (with radius

[1] This work has been supported by the Chinese National Key Basic Research (973) Projects 2004CB318000 and 2005CB321901/2 and the SKLSDE Project 07-003.

[2] Email: Dongming.Wang@lip6.fr

0)? These simple questions cannot be easily answered. This is partially because some geometric concepts are introduced to model the real world informally with vague descriptions and their definitions may never be used in formal reasoning. We need to distinguish such concepts from those that can be formally defined. Even though most of the geometric objects and relations in standard textbooks are defined formally, the rigorousness of their definitions still remains questionable (because of the customary ignorance of degeneracy). Therefore, formalizing and specifying (symbolic) geometric objects and relations are fundamental issues that have to be considered and studied carefully for the representation and manipulation of such objects on computer. Our work on the identification, formalization, structuring, and specification of geometric knowledge objects has been motivated by our design and implementation of a dynamic geometry software environment with formalized knowledge base [1].

What is geometric knowledge? To be specific, we restrict ourselves to elementary geometry with plane Euclidean geometry as a concrete example. By *geometric knowledge* we mean the totality of *knowledge objects* including

(1) concepts, theorems, proofs, problems, solutions, diagrams, explanations, etc. that exist in the literature of geometry;

(2) methods/algorithms, techniques, strategies, rules, heuristics, etc. that have been introduced and developed to define geometric concepts, to prove geometric theorems, to solve geometric problems, and to draw geometric diagrams;

(3) relations among knowledge objects (1)–(2);

(4) methods and techniques for managing knowledge objects (1)–(2) and knowledge object relations (3).

These knowledge objects have been introduced and presented with certain structure in formal or informal publications using natural mathematical languages along with the development of geometry. Roughly speaking, knowledge objects of types (1) and (3) can be stored electronically as data in knowledge bases, while knowledge objects of types (2) and (4) can be implemented as procedures in software modules. The procedures may process data and the data may contain instructions for invoking procedures. Our main objective here is to standardize, formalize, and specify knowledge objects of type (1) and to investigate the formalization and specification of knowledge objects of types (2) and (3). For this, it is necessary to identify such objects according to their features and roles in geometry.

The management of geometric knowledge amounts to processing geometric knowledge objects, such as organizing knowledge data, presenting knowledge data in human-readable format, modifying and restructuring presentations of knowledge data, invoking implemented algorithms, techniques, strategies, and heuristics for geometric computing, reasoning, and drawing with knowledge data, and manipulating produced knowledge objects (geometric objects and relations, proofs, diagrams, etc.). The reader may refer to [4] for the current state of the art of mathematical knowledge management in general.

2 Formalization and Structuring of Geometric Knowledge Objects

The first geometric knowledge object we need to consider is *Geometric Concept*. A geometric concept is either a *geometric object*, or a *geometric quantity*, or a *geometric relation*. More formally, we define it as

$$\text{geoConcept} := \text{geoObject} \mid \text{geoQuantity} \mid \text{geoRelation}$$

where | stands for "or" (with & for "and") and geoObject is either a

- *basic geometric object* (like point or circle) without formal definition, or a
- *derived geometric object* (like midpoint or parallelogram) formally defined by a function from other geometric objects,

geoQuantity is a pair

- $\langle q, u \rangle$ with q being an algebraic quantity (usually real) and u a geometric unit (like length, area, or degree) formally defined by a function from geometric objects,

and geoRelation is a relation

- (like parallelism or collinearity) among geometric objects, or
- (like "=" or ">") among geometric quantities

formally defined by a predicate from geometric objects or from geometric quantities and it has a truth value (say `true` by default).

Basic geometric objects are introduced informally, i.e., by using informal descriptions in natural languages, as primitive elements for the construction of geometry. For instance, one can take points, lines, angles, triangles, and circles as basic geometric objects for plane Euclidean geometry. We leave aside such questions as which geometric objects are *really basic* and whether the chosen set of basic geometric objects is *minimal*.

Having a (small) set of basic geometric objects fixed, new geometric objects, quantities, and relations may be introduced formally and constructively by definitions. So *Definition* is another knowledge object that we need. A definition is used to define a new geometric concept, possibly with some *associated geometric concepts*, formally from basic geometric objects or already defined geometric concepts. It has the following form.

Let $O_1, \ldots, O_t$ be basic geometric objects or already defined geometric objects or quantities and $R_1, \ldots, R_v$ be already defined geometric relations among $O_1, \ldots, O_t$.

Definition.

(1) An already defined geometric object O is called the/a *new geometric object* of $O_1, \ldots, O_t$, denoted as $O := \text{newObject}(O_1, \ldots, O_t)$, if the geometric relations $R_1, \ldots, R_v$ are satisfied.

(2) The *new geometric quantity* $Q = \langle q, u \rangle$ of $O_1, \ldots, O_t$, denoted as $Q := \text{newQuantity}(O_1, \ldots, O_t)$, is a function of $O_1, \ldots, O_t$, where q is an algebraic quantity computable from $O_1, \ldots, O_t$ after algebraization and u is a geometric unit.

(3) The geometric objects or quantities $O_1, \ldots, O_t$ are said to satisfy the *new geometric relation* R, denoted as $R(O_1, \ldots, O_t)$, if the geometric relations $R_1, \ldots, R_v$ are satisfied.

For example, let `A`, `B`, `C` be points, $\texttt{on}(\texttt{C}, \texttt{line}(\texttt{A}, \texttt{B}))$ stand for "point `C` is on line `AB`," and $\texttt{distance}(\texttt{A}, \texttt{B})$ denote the distance between the two points `A` and `B`. Assume that the geometric relation `on` and the geometric quantity `distance` have already been defined. Then

- A point `C` is called the *midpoint* of `A` and `B`, denoted as $\texttt{C} := \texttt{midpoint}(\texttt{A}, \texttt{B})$, if $\texttt{on}(\texttt{C}, \texttt{line}(\texttt{A}, \texttt{B}))$ & $\texttt{distance}(\texttt{A}, \texttt{C}) = \texttt{distance}(\texttt{C}, \texttt{B})$.
- The *perimeter* $\langle q, u \rangle$ of triangle `ABC`, denoted as
$$\langle q, u \rangle := \texttt{perimeter}(\texttt{triangle}(\texttt{A}, \texttt{B}, \texttt{C})),$$
is a function of `A`, `B`, `C` with $q := \texttt{distance}(\texttt{A}, \texttt{B}) + \texttt{distance}(\texttt{B}, \texttt{C}) + \texttt{distance}(\texttt{C}, \texttt{A})$ and u being the geometric unit `Length`.
- A triangle `ABC` is said to be *isosceles*, denoted as $\texttt{isosceles}(\texttt{triangle}(\texttt{A}, \texttt{B}, \texttt{C}))$, if $\texttt{distance}(\texttt{A}, \texttt{B}) = \texttt{distance}(\texttt{B}, \texttt{C})$ | $\texttt{distance}(\texttt{B}, \texttt{C}) = \texttt{distance}(\texttt{C}, \texttt{A})$ | $\texttt{distance}(\texttt{C}, \texttt{A}) = \texttt{distance}(\texttt{A}, \texttt{B})$.

After geometric objects, quantities, and relations have been introduced gradually, one of the most important tasks in geometry is to study logical relations among geometric relations. So now we come to the knowledge object *Geometric Theorem*. A geometric theorem is a statement about some logical relations among geometric relations that has already been proved to be true logically. It may be formulated in the following typical form.

Theorem. Let $O_1, \ldots, O_t$ be geometric objects or geometric quantities and $R_1, \ldots, R_v$ be geometric relations among $O_1, \ldots, O_t$. For all $O_1, \ldots, O_s$ $(s \leq t)$, there exist $O_{s+1}, \ldots, O_t$ such that,
$$\text{if } R_1 \,\&\, \cdots \,\&\, R_u \ (u < v) \text{ holds, then } R_{u+1} \,|\, \cdots \,|\, R_v \text{ holds.}$$

As usual, $R_1 \,\&\, \cdots \,\&\, R_u$ is called the *hypothesis* and $R_{u+1} \,|\, \cdots \,|\, R_v$ the *conclusion* of the theorem. For most theorems in elementary geometry, $s = t$ and $v = u + 1$, so no existential quantifier and disjunction are involved. If the conclusion of a theorem is a conjunction of several geometric relations or the hypothesis involves disjunctions, one may split the theorem into several ones, so that there is only one geometric relation in the conclusion and no disjunction in the hypothesis of each split theorem. Of course, there are geometric theorems with more complex structure that cannot be put in the above form.

There are several types of statements in geometry, or mathematics in general, which are similar or closely related to theorems. They include axioms, propositions, lemmas, corollaries, and conjectures, which are all about logical relations among geometric relations and have the same form as theorems. Let us distinguish them from theorems as follows. The truth of an axiom is assumed on the basis of beliefs, observations, or experiments without formal proof. A proposition may be true or false (but only true propositions are given usually). A proved true statement is a lemma only if it is used in the proof of at least one theorem or another lemma. A

true statement is a corollary only if it follows directly from one or more theorems or lemmas without any need of proof. The truth of a conjecture is unknown but is expected to be proved.

Some of these distinctions may be arguable and need to be justified, but this is not really essential. What is essential in our approach is the emphasis on specifying mathematical objects, relations, terms, etc. explicitly, formally, and unambiguously. It is also for this reason that we shall intentionally not allow a point to be considered as a special, degenerated segment, triangle, or circle (see the specifications in Section 4).

Naturally, we need the knowledge object *Proof*. Our attempt is to identify, formalize, and specify most of the standard knowledge objects of geometry, including *Problem*, *Example*, *Exercise*, and *Solution*, where *Example* and *Exercise* may be considered as some kind of *Problem*. We expect that such knowledge objects will cover the main contents of geometry, though they cannot be exhaustive. The abstracted entities and their relationships for the Geometric Concept object and other knowledge objects are presented in [1] by using entity-relationship diagrams for the design of a geometric knowledge base.

The above analysis also shows the hierarchic structure of geometric knowledge objects. We sketch this structure by the following diagram.

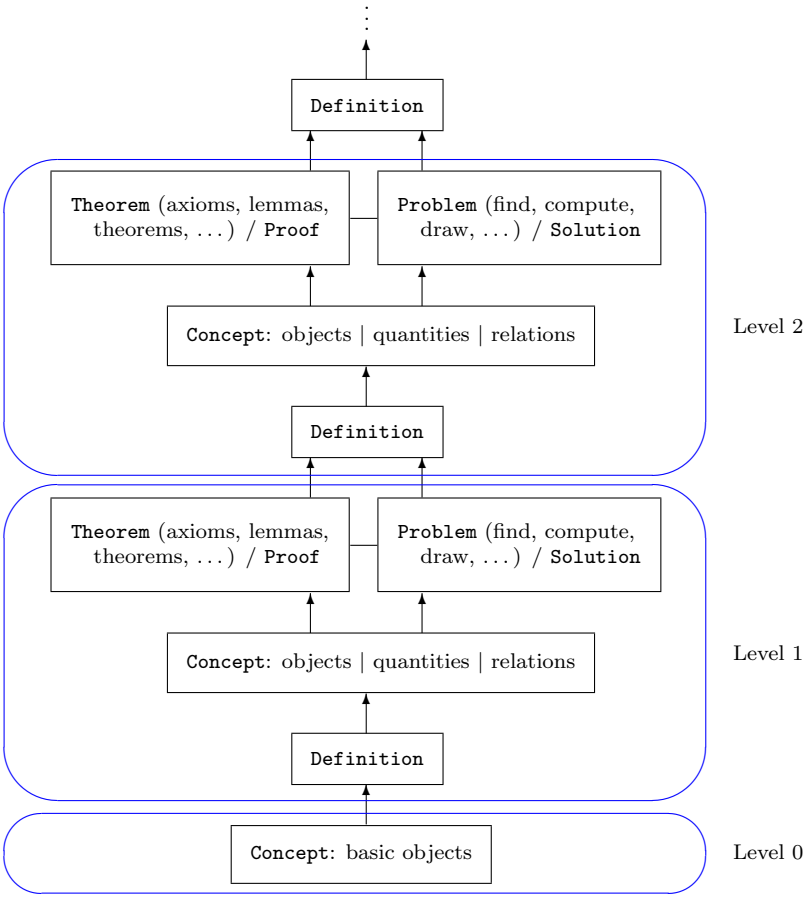


Fig. 1. Structure of Geometric Knowledge

3 Predicates and Functions with Embedded Knowledge

The representations such as `midpoint(A,B)`, `distance(A,B)`, and `on(C,line(A,B))` used in the preceding section originate from predicate logic [5]. Note that `midpoint`, `distance`, and `line` are function symbols denoting geometric objects or quantities, whereas `on` is a predicate symbol denoting a geometric relation. These predicate and function notations are standard and have been used in different domains.

While describing human relationships in the language of predicate logic, one may use, e.g., the predicate `father(F,C)` to denote "`F` is the father of `C`." In view of the circumstances, we know that both `F` and `C` are human beings and `C` is a son or a daughter of `F`. However, this implied information is known only to us, the users of the predicate langauge, but not associated with the predicate. Thus the information cannot be retrieved and used in the process of reasoning with the predicate by computer programs. Current inference systems for predicate logic are designed mostly at the level of purely logical reasoning without using any knowledge from specific domains. For encoding domain-specific knowledge, one may need to use sorted or higher-order logic (see, e.g., [3]). We propose to embed some of the knowledge (structured or unstructured) from the domain in question into predicates and functions using a simple mechanism, so that such domain knowledge can be used, whenever needed, to increase the power and performance of automated processing and reasoning. This approach is particularly effective for the management of geometric knowledge using predicate logic because the amount of information implied in each predicate can be very large.

Let $P(x_1,\ldots,x_n)$ be an n-ary predicate or function in a first-order language with structure $\mathbf{S}$. $\mathbf{S}$ consists of a domain $\mathbb{D}$ (e.g., Euclidean geometry) and an interpretation, by which $x_1,\ldots,x_n$ and $P(x_1,\ldots,x_n)$ can be interpreted meaningfully in $\mathbb{D}$. Let $\mathbf{v}_1,\ldots,\mathbf{v}_t$ be t segments of knowledge about $x_1,\ldots,x_n$ with respect to $P(x_1,\ldots,x_n)$ in $\mathbb{D}$. Assign a key k_i to each $\mathbf{v}_i$ and create a (hash) table

$$K := \left\{ \begin{array}{c} k_1(x_1,\ldots,x_n) = \mathbf{v}_1; \\ k_2(x_1,\ldots,x_n) = \mathbf{v}_2; \\ \cdots\cdots \\ k_t(x_1,\ldots,x_n) = \mathbf{v}_t \end{array} \right\}.$$

We embed the table K into $P(x_1,\ldots,x_n)$ and call the result a *predicate* or *function with embedded knowledge* and each entry $k_i(x_1,\ldots,x_n) = \mathbf{v}_i$ of K a *knowledge entry* with *key* k_i and *value* $\mathbf{v}_i$. $P(x_1,\ldots,x_n)$ functions as it is in first-order logic and the table K is hidden. The embedding of K into $P(x_1,\ldots,x_n)$ can be easily implemented at the level of programming. We devise the following simple mechanism to retrieve knowledge keys and values:

- $P[\,](x_1,\ldots,x_n)$ returns the sequence of knowledge keys $k_1,\ldots,k_t$;
- $P[k_{i_1},\ldots,k_{i_r}](y_1,\ldots,y_n)$ returns the sequence of knowledge values

$$\mathbf{v}_{i_1}|_{x_1=y_1,\ldots,x_n=y_n}, \ldots, \mathbf{v}_{i_r}|_{x_1=y_1,\ldots,x_n=y_n},$$

where $\mathbf{v}_{i_j}|_{x_1=y_1,\ldots,x_n=y_n} = k_{i_j}(y_1,\ldots,y_n)$.

The knowledge values of K may contain other predicates or functions with embedded knowledge.

As an example, we use the predicate $\texttt{on}(\texttt{C},\texttt{line}(\texttt{A},\texttt{B}))$ to illustrate the embedding and retrieving of knowledge keys and values. In classical mathematical logic, this predicate denotes only a relation between C and $\texttt{line}(\texttt{A},\texttt{B})$. However, to the user of the predicate representation there is a lot of other information associated with this predicate. We can embed part of the information into the predicate by creating, e.g., the following table:

$$\left\{\begin{array}{l}
\texttt{typeOf}(\texttt{C},\texttt{line}(\texttt{A},\texttt{B})) = [\texttt{Point},\texttt{Line}];\\
\texttt{meaningEnglish}(\texttt{C},\texttt{line}(\texttt{A},\texttt{B}))\\
\quad = \texttt{the point C is on line}[\texttt{meaningEnglish}](\texttt{A},\texttt{B});\\
\texttt{meaningEnglishNegation}(\texttt{C},\texttt{line}(\texttt{A},\texttt{B}))\\
\quad = \texttt{the point C is not on line}[\texttt{meaningEnglish}](\texttt{A},\texttt{B});\\
\texttt{algebraicExpression}(\texttt{C},\texttt{line}(\texttt{A},\texttt{B}))\\
\quad = [\texttt{A}_1\texttt{B}_2 + \texttt{A}_2\texttt{C}_1 + \texttt{B}_1\texttt{C}_2 - \texttt{A}_1\texttt{C}_2 - \texttt{A}_2\texttt{B}_1 - \texttt{B}_2\texttt{C}_1 = 0];\\
\texttt{drawingInstruction}(\texttt{C},\texttt{line}(\texttt{A},\texttt{B})) = [\texttt{line}(\texttt{A},\texttt{B}),\texttt{line}(\texttt{A},\texttt{C})];\\
\texttt{degeneracyCondition}(\texttt{C},\texttt{line}(\texttt{A},\texttt{B})) = [\texttt{coincide}(\texttt{A},\texttt{B})]
\end{array}\right\},$$

in which $\texttt{A}_\texttt{i} = \texttt{algebraicExpression}[\texttt{i}](\texttt{A})$ denotes the ith coordinate of point A, and similarly for $\texttt{B}_\texttt{i}$ and $\texttt{C}_\texttt{i}$. Let $\texttt{line}[\texttt{meaningEnglish}](\texttt{A},\texttt{B})$ return "the line AB." Then

- $\texttt{on}[\texttt{meaningEnglish}](\texttt{R},\texttt{line}(\texttt{P},\texttt{Q}))$ yields "the point R is on the line PQ," the meaning of the predicate $\texttt{on}(\texttt{R},\texttt{line}(\texttt{P},\texttt{Q}))$ stated in English, and
- $\texttt{on}[\texttt{algebraicExpression}]([\texttt{c},\texttt{d}],\texttt{line}([\texttt{a},\texttt{b}],[0,2]))$ yields $[\texttt{bc}+2\,\texttt{a}-\texttt{ad}-2\,\texttt{c}=0]$, the algebraic expression of $\texttt{on}([\texttt{c},\texttt{d}],\texttt{line}([\texttt{a},\texttt{b}],[0,2]))$.

These values of the embedded knowledge can be used to compose statements in natural languages and to produce algebraic expressions for computing and reasoning, while the knowledge values linked to the keys drawingInstruction and degeneracyCondition may be used to draw diagrams and to generate nondegeneracy conditions respectively and automatically with the predicate.

4 Formal Specification of Geometric Knowledge Objects

As mentioned before, we choose a small number of geometric objects as basic ones. Consider for instance plane Euclidean geometry and let points, lines, angles, polygons, and circles be chosen as basic geometric objects, which are defined by informal descriptions in natural mathematical languages and have respective types Point,

`Line`, `Angle`, `Polygon`, and `Circle`. Moreover, we use type `Real` for geometric quantities and type `Boolean` for geometric relations. These seven types are basic root types which have no father type. Basic geometric objects and initially defined geometric quantities (like `distance`) and relations (like `coincide`) of these seven types are called *geometric primitives*. Our approach proceeds by constructing other geometric objects, quantities, and relations (i.e., instances of geoConcept) formally and successively from geometric primitives. Meanwhile, geometric theorems (as well as axioms, propositions, lemmas, etc. about geometric properties of constructed objects, quantities, and relations) and their proofs, geometric problems (as well as examples and exercises about deriving geometric properties, computing geometric quantities, and drawing geometric diagrams) and their solutions, etc. are formulated (completely or partially) by using formal languages. We refer to the process of construction and formulation as the *formalization* of geometric knowledge objects. When formalized and specified, geometric knowledge objects can be stored electronically and structurally as patterns with semantics and the management of geometric knowledge can be mechanized and automated.

To formally specify knowledge objects of Euclidean geometry, let us first define a *point*, denoted by a capital letter like `P`, informally as *an entity that has a location in the space or on a plane, but has no extent* (or as "*that which has no part*" according to Euclid's original vague definition). It has neither volume, area, length, nor any other higher dimensional analogue. A *line* is defined informally as *a path of one point moving straightly* (or "*breadthless length*" according to Euclid). A line has (infinite) length but no width. For any two distinct points `A` and `B`, there is a *line* passing through these two points. We denote this line by `AB`. A (line) *segment* connecting `A` and `B`, denoted by $\overline{\texttt{AB}}$, is part of the line `AB` that is between `A` and `B`. It has a finite length. We will define *segment* formally.

Using the structure of geometric knowledge objects we have analyzed and predicates and functions with embedded knowledge, we can readily specify other geometric concepts by formal definitions and geometric theorems by formal statements. In the following formal specifications, `::` stands for "assumed to be (of type)," `*` for multiplication, `^` for power, `l[i]` for the `i`th element of a list (or an ordered set) `l`, and $\overline{\texttt{AB}}$ for `segment(A, B)` in formatted mathematical text.

```
Definition geoQuantity <d, l> := distance(A, B) {
  let d::Real = |AB| & l::Length;
  knowledgeEmbedded {
    typeOf(A, B) = [Point, Point];
    meaningEnglish(A, B) = the distance between the points A and B;
    property(A, B) = [d ≥ 0];
    algebraicExpression(A, B) = [sqrt(
      (algebraicExpression[1](A) - algebraicExpression[1](B))^2 +
      (algebraicExpression[2](A) - algebraicExpression[2](B))^2 )]
    }
  }
```

```
Definition geoObject s := segment(A, B) {
  let s::Line = AB;
  ifCondition::Boolean distance(A, B) > 0::true;
  associatedConcept {
    geoQuantity <l, u> := length(s) {
      let l::Real = |s| & u::Length;
      knowledgeEmbedded {
        meaningEnglish(s) = the length of s;
        property(s) = [l > 0];
        algebraicExpression(s) = [distance(A, B)]
      }
    }
  }
  knowledgeEmbedded {
    typeOf(A, B) = [Point, Point];
    meaningEnglish(A, B) = the segment AB;
    algebraicExpression(A, B) =
      [(1-t)*algebraicExpression[i](A)+t*algebraicExpression[i](B))/2
        $ i = 1..2, 0 ≤ t ≤ 1];
    drawingInstruction = [segment(A, B)]
  }
}

Definition geoRelation collinear(A, B, C) {
  ifCondition::Boolean
    coincide(A, B)::true | on(C, line(A, B))::true;
  knowledgeEmbedded {
    typeOf(A, B, C) = [Point, Point, Point];
    meaningEnglish(A, B, C)
      = the three points A, B and C are collinear;
    meaningEnglishNegation(A, B, C)
      = the three points A, B and C are not collinear;
    algebraicExpression(A, B, C)
      = on[algebraicExpression](C, line(A, B));
    drawingInstruction = [line(A, B), line(A, C)]
  }
}
```

In the above it is assumed that on(C, line(A, B)) and the trivial geometric relation coincide(A, B), meaning that "the two points A and B coincide," have already been specified. Similarly, we can specify

- the trivial relation arbitrary(x, y, z, ...), meaning that "x, y, z, ... are/is arbitrary,"

- the geometric object `circumcircle(triangle(A, B, C))`, that is, "`the circumcircle of the triangle ABC`," and
- the geometric relation `perpendicularFoot(P, D, line(A, B))`, meaning that "`the point P is the foot of the perpendicular drawn from the point D to the line AB`."

A formal specification of `triangle(A, B, C)`, the triangle `ABC` formed with three points `A`, `B` and `C`, as a derived geometric object of `Polygon`, is given in the appendix. Now we specify Simson's theorem formally as follows.

```
Theorem Simson := theorem(h, c) {
  hypothesis h = [arbitrary(triangle(A, B, C)),
    on(D, circumcircle(triangle(A, B, C))),
    perpendicularFoot(P, D, line(A, B)),
    perpendicularFoot(Q, D, line(A, C)),
    perpendicularFoot(R, D, line(B, C))];
  conclusion c = [collinear(P, Q, R)];
  knowledgeEmbedded {
    proof(h, c) = [...];
    lemma(h, c) = [...];
    corollary(h, c) = [...];
    reference(h, c)
      = [H.S.M. Coxeter & S.L. Greitzer: Geometry Revisited, p. 41];
    keyword = [Simson line, Wallace, ...];
    derivedConcept(h, c) = [SimsonLine(D, triangle(A, B, C)), ...];
    note(h, c) = [Robert Simson (1687-1768) made several contributions
      to both geometry and ...  The 'simson' was attributed to him
      because it seemed to be typical of his geometrical ideas ...
      Actually it was discovered in 1797 by William Wallace.];
        ......
  }
}
```

As the predicates and functions involved in Simson's theorem have embedded knowledge, the hypothesis and conclusion in the above specification contain a remarkable amount of information that can be used for processing the theorem, such as automatically

- translating the specification of the theorem into an English statement, into a first-order logical formula, or into algebraic expressions,
- drawing animatable diagrams for the theorem,
- proving the theorem using algebraic methods, and
- generating nondegeneracy conditions in geometric form

(cf. [6]). In fact, we have used predicates and functions with embedded knowledge

for the specification of geometric theorems since the early 1990s. It is this formalism that makes our prover GEOTHER [6] capable of handling geometric theorems automatically. The formalism of predicates and functions with embedded knowledge has been further used for the creation of electronic and dynamic geometric documents [2] and the design and implementation of a geometric knowledge base [1].

In the specification of geometric theorems, `hypothesis` and `conclusion` are two key attributes. Characteristic attributes may also be identified for other geometric knowledge objects: for instance, `given`, `find`, `compute`, and `draw` for geometric problems. We may structure human proofs of theorems and solutions of problems into steps and substeps, and formalize logical reasoning steps using first-order logic and computational steps by means of algorithms. In this way, we will be able to formally specify a large portion of geometric knowledge.

Of course, there are informal explanation texts written in natural languages that cannot be easily formalized. The treatment of such mathematical `Text` objects is beyond the scope of our present study.

5 Final Remarks

Our investigations on the formalization and specification of geometric knowledge objects have been carried out primarily for the purpose of creating a formalized geometric knowledge base. With such a knowledge base, we can develop a geometry software environment (Fig. 2) in which geometric computing, reasoning, and drawing modules and knowledge management tools are integrated.

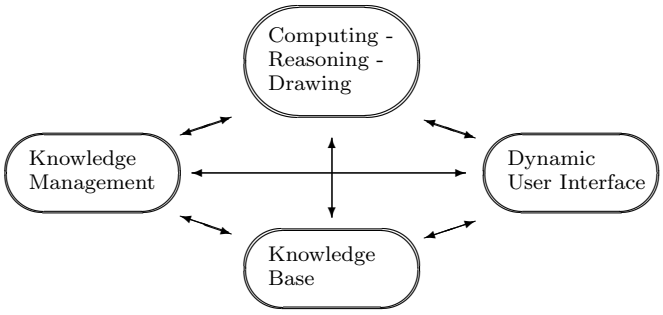


Fig. 2. Geometry Software Environment

When knowledge objects (algorithms, techniques, strategies, rules, heuristics, etc.) of types (2) and (4) listed in Section 1 are efficiently implemented, we can mechanize and automate the process of geometric computing, reasoning, drawing, and knowledge managing to a considerable extent, as shown by some of the functionalities of our primitive systems [1,2,6].

References

[1] X. Chen, Y. Huang, and D. Wang: On the Design and Implementation of a Geometric Knowledge Base. In: *Proceedings of ADG 2008 — Seventh International Workshop on Automated Deduction in Geometry* (Shanghai, China, September 22–24, 2008) (M. Kauers, M. Wu, and Z. Zeng, eds.), East China Normal University, China, 2008, pp. 62–78. Chinese version in *Journal of Computer Applications* (to appear).

[2] X. Chen and D. Wang: Towards an Electronic Geometry Textbook. In: *Automated Deduction in Geometry* (F. Botana and T. Recio, eds.), LNAI **4869**, pp. 1–23. Springer-Verlag, Berlin Heidelberg (2007).

[3] M. Kerber: *On the Representation of Mathematical Concepts and their Translation into First-Order Logic*. Ph.D. thesis, Fachbereich Informatik, Universität Kaiserslautern (1992).

[4] M. Kerber (ed.): *Management of Mathematical Knowledge*. Special issue of *Mathematics in Computer Science*, vol. 2, no. 2. Birkhäuser, Basel (2008).

[5] W. Li: *Mathematical Logic: Basic Principles and Formal Calculus*. Birkhäuser, Basel (2009, to appear). Chinese edition: Science Press, Beijing (2007).

[6] D. Wang: GEOTHER 1.1: Handling and Proving Geometric Theorems Automatically. In: *Automated Deduction in Geometry* (F. Winkler, ed.), LNAI **2930**, pp. 194–215. Springer-Verlag, Berlin Heidelberg (2004).

Appendix. Specification of a General Triangle

```
Definition geoObject t := triangle(A, B, C) {
  let t::Polygon = ΔABC;
  ifCondition::Boolean collinear(A, B, C)::false;
  associatedConcept {
    geoObject v := vertex(t) {
      let v::[Point] = [A, B, C];
      knowledgeEmbedded {
      meaningEnglish(t) = the vertices A, B and C of t
      }
    }
    geoObject s := side(t) {
      let s::[Line] = [segment(A,B), segment(B,C), segment(C,A)];
      knowledgeEmbedded {
        meaningEnglish(t) = the sides AB, BC and CA of t;
        property(t) = [length(s[i]) + length(s[j]) > length(s[k])
          $ i≠j≠k, i = 1..3]
      }
    }
    geoObject a := angle(t) {
      let a::[Angle] = [angle(C,A,B),angle(A,B,C),angle(B,C,A)];
      knowledgeEmbedded {
        meaningEnglish(t) = the angles ∠CAB, ∠ABC and ∠BCA of t;
        property(t) = [degree(a[1]) + degree(a[2]) + degree(a[3])
          = <π, Degree>]
      }
    }
    geoQuantity <a, u> := area(t) {
      let a::Real & u::Area;
      knowledgeEmbedded {
```

```
        meaningEnglish(t) = the area of t;
        property(t) = [a > 0];
        algebraicExpression(s)
          = [distance(A, B)*distance(C, line(A, B))/2]
      }
    }
    geoQuantity <p, l> := perimeter(t) {
      let p::Real & l::Length;
      knowledgeEmbedded {
        meaningEnglish(t) = the perimeter of t;
        property(t) = [p > 0];
        algebraicExpression(s)
          = [distance(A, B) + distace(B, C) + distace(C, A)]
      }
    }
  knowledgeEmbedded {
    typeOf(A, B, C) = [Point, Point, Point];
    meaningEnglish(A, B, C) = the triangle ABC;
    derivedConcept(A, B, C) = [centroid(t), orthocenter(t),
      circumcenter(t), incenter(t), escenter(t), circumcircle(t),
      inscribedCircle(t), escribedCircle(t)];
    drawingInstruction = [segment(A,B),segment(B,C),segment(C,A)];
    degeneracyCondition = [collinear(A, B, C)]
  }
}
```

A Revised Pointer Logic for Verification of Pointer Programs

Zhaopeng Li [†⋆], Yiyun Chen [†⋆], Baojian Hua [‡] and Zhifang Wang [†⋆1]

† *Department of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui 230026, China*

⋆ *Suzhou Institue for Advanced Study, University of Science and Technology of China, Suzhou, Jiangsu 215123, China*

‡ *School of Software Engineering, University of Science and Technology of China, Hefei, Anhui 230026, China*

Abstract

Proof-Carrying Code brings two grand challenges to the research field of programming languages. One is to seek more expressive type or logic systems to specify or reason about the properties of high-level or low-level programs. This paper improves and extends the pointer logic we designed before for verifying C-like pointer programs.
The main contribution is that we present a concept of legal sets of access paths, simplify elementary operations on access paths, and make inference rules easy to understand. Furthermore, we extend the logic with inference rules for local reasoning and function constructs, make the logic conveniently used in the context of function calls. We implement a core part of pointer logic in Calculus of Inductive Constructions. Finally, we give a comprehensive comparison between pointer logic and separation logic. The pointer logic exceeds in ruling out memory leaks by syntactic method.

Keywords: Software Safety, Hoare Logic, Pointer Logic,Proof-Carrying Code, Certifying Compiler

1 Introduction

Proof-Carrying Code (PCC) [1], as a new code paradigm, brings two grand challenges to the research field of programming languages. One is to seek more expressive logics or type systems to specify or reason about the properties of high-level or low-level programs. The other is to study the technology of certifying compilation in which the compiler generates proofs automatically for programs with annotations.

For the first challenge, Typed Assembly Language (TAL) [2] and Type Refinements [3] are two typical research projects using type-based approaches. While PCC and Certified Assembly Programming (CAP) [4,5] are typical research projects on logic-based techniques. Type-based and logic-based techniques are complementary

[1] Email:{zpli,huabj,zfwang7}@mail.ustc.edu.cn yiyun@ustc.edu.cn

to each other, and by combining these techniques, Applied Type System (ATS) project [6] proposed by Xi *et al.* extends the type system with a notion of program states. So by encoding Hoare logic in its type system, ATS can support Hoare-logic-like reasoning via the type system.

For the second challenge, Necula pioneered a certifying compiler [7] called Touchstone. It contains a traditional compiler for a small but type-safe subset of C-like language and a certifier that automatically produces a proof of type safety for each assembly program generated by the compiler. The major drawback of Touchstone are in the aspects of pointer types and dynamic storage allocation.

Recently, we have studied methods to apply techniques of proof-carrying code and certifying compilation to a programming language with explicit memory management. We have designed the PointerC language [8] which is a C-like language with dynamic memory allocation and deallocation. The elementary safety policy is that there are no operations such as dereference and free on null pointers or dangling pointers, no memory leaks during the program execution and etc.. To reason about such properties of the program, we adopt a method combining techniques of type and logic systems. In order to design a simple yet sound type system, we introduce side conditions in the typing rules which makes restrictions on the value of the syntactic expressions. To check these side conditions, a pointer logic [9] has been designed for PointerC. The pointer logic is an extension of Hoare logic. It is used to deduce the precise pointer information at each program point such as whether a pointer is null, dangling, or valid (a valid pointer points to an object in the heap) and the equality between valid pointers. All information is used to prove whether pointer programs satisfy the side conditions, thus it can support safety verification of pointer programs. We have proved the safety theorem of PointerC and the soundness theorem of the pointer logic[10] using the proof assistant Coq [11]. Furthermore, we have implemented a certifying compiler prototype for PointerC [8,9].

This paper improves and summarizes the design of the pointer logic. The main contributions are:

- We propose a concept, namely *legal access path set*, to express pointer information at program points concisely. Therefore we redesign the pointer logic in order to make the inference rules easy to understand. In essence, pointer logic is a precise pointer analysis tool, but it uses Hoare triple to represent the process of obtaining the precise pointer information and use such information in Hoare-style program verification directly.
- We implement a core part of pointer logic in Calculus of Inductive Constructions (CiC) [11]. The implementation can be regarded as a verification tool of pointer programs at source level. It can be used to prove safety properties of programs manipulating complex data structures in which pointer equality is in a regular pattern such as single linked list, circular doubly-linked list and binary tree.
- We give an in-depth comparison between pointer logic and separation logic. Pointer logic corresponds to separation logic for high level languages. But the pointer logic rules out memory leaks by syntactic method, while separation logic can rule it out by restricting the assertion to precise one.

$$
\begin{aligned}
\mathit{type} &::= \texttt{bool} \mid \texttt{int} \mid \texttt{struct}\ \mathit{ident}* \\
\mathit{stmtlist} &::= \mathit{stmtlist}\ \mathit{stmt} \mid \mathit{stmt} \\
\mathit{stmt} &::= \mathit{lval}\ \texttt{=}\ \mathit{exp};\ \mid \mathit{lval}\ \texttt{=}\ \mathit{alloc}(\texttt{struct}\ \mathit{ident});\ \mid \texttt{free}(\mathit{exp}); \\
&\mid\ \texttt{if}(\mathit{boolexp})\ \mathit{block}\ \texttt{else}\ \mathit{block};\ \mid \texttt{while}(\mathit{boolexp})\ \mathit{block}; \\
\mathit{block} &::= \mathit{stmt} \mid \{\mathit{stmtlist}\} \\
\mathit{lval} &::= \mathit{ident} \mid \mathit{lval}\texttt{->}\mathit{ident}
\end{aligned}
$$

Fig. 1. Representative Syntax of the PointerC Language

This paper presents a introduction to the redesigned pointer logic. Limited space, some details are given in an extended version of this paper[12]. The rest of the paper is organized as follows. PointerC is briefly introduced in section 2. In section 3 we present the re-designed pointer logic. In section 4 we introduce the implementation of this logic system using the proof assistant Coq. A case study is given in section 5. We do a comparison between pointer logic and separation logic in section 6. Section 7 compares our work with related works and section 8 concludes.

2 PointerC Language

PointerC is a C-like programming language in which the pointer type is emphasized (for details of type system, see[13]). And the representative syntax is shown in Figure 1. In PointerC, pointer variables can only be used in assignment statements, equality test expressions, in operations like storing and loading the value which they are pointing to, and as parameters of functions. Pointer arithmetic and the address-of operator (&) are forbidden. Functions `malloc` and `free` are regarded as pre-defined functions in PointerC which satisfy the minimal safety policy. For example, every call to `malloc` always succeeds and the allocated heap spaces will never overlap. In addition, short-circuit calculation is not adopted in evaluation of PointerC boolean expressions so that PointerC boolean expressions can be used directly in assertions. Such consideration is made since short-circuited *and* operation and *or* operation are not exchangeable operations when pointers appear. In this paper, *lval*s starting with declared pointer variables are named *access paths*. And the prefixes of one access path only include the prefixes which are also access paths. Similarly, adding a suffix to one access path only considers that the result of this operation is also an access path. For example, assume that `p` is an access path which points to:

```
struct node {int data; struct node *l, *r},
```

then prefixes of access path `p->l->r` are `p->l` and `p`. And the result access path by adding suffixes to `p` are `p->l` and `p->r`. Access paths of pointer types are called pointers in this paper.

At any program point, we use $\mathcal{N}$ and $\mathcal{D}$ to denote a set of pointer access paths respectively. And we use Π to denote a set of pointer access path set. All access paths in the same subset of Π have the same type. And no access path appears

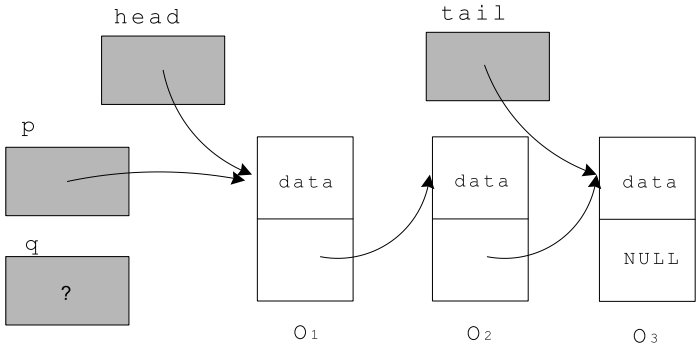


Fig. 2. An Example of Ψ

more than once in Π. Ψ is used to denote $\mathcal{N}$, $\mathcal{D}$ and Π in the following text. On the semantic model of PointerC, $\mathcal{N}$ is `NULL` pointer set and $\mathcal{D}$ is a set of dangling pointers. Pointers in one set of Π are pointing the same object in the heap (that is, they have the same *rvalue*) and pointers in different sets of Π are pointing to different objects. Pointers in Π are called *valid pointers*. Therefore, Ψ is an abstraction of pointer relations at one program point where equality is emphasized, but concrete values of the pointers are not concerned. Pointers in $\mathcal{N}$ are differentiated from the ones in $\mathcal{D}$ because `NULL` pointers can be distinguished from dangling pointers by testing the equality of pointers and `NULL` in programs.

For example, there is a singly-linked list with three nodes in Figure 2. The type of the node is:

```
struct List {int data; struct List *next}.
```

`head, tail, p` and `q` are four pointers. Note that `q` is a dangling pointer. Obviously `tail->next` is `NULL` pointer, so $\mathcal{N}$ is {`tail->next`}. And $\mathcal{D}$ is {`q`} as aforementioned. Variables `head` and `p` are pointing to the head of the list (object O_1 in the heap) and `tail` is pointing to the tail (object O_3). The second node (object O_2) is pointed by `p->next` and `head->next`. In this case Π includes:

```
{head, p},{p->next},{p->next->next, tail},
```

in which the three sets stand for object O_1, O_2 and O_3 in the heap respectively. We only preserve concise pointer information here. It is worth noting that `head->next` is not included in the second set of Π because it can be figured out using the first set. Set {`head, p`} indicates that both `head` and `p` point to the same object, so `head->next` and `p->next` are equal. In such a case, we use only one of them as the access path pointing to O_2. And `head->next->next` is excluded from the third set for a similar reason.

Since not arbitrary Ψ can express the mentioned meaning, we need to define *legal* Ψ. First, we define alias of access path based on Π. Result access paths by adding same suffix to pointers in a same set of Π are aliases. And alias relation of access path is reflexive. By this definition, the predicate judging whether two access paths are aliases are as follow:

$$\begin{aligned}\mathsf{alias(p,q)} \triangleq \mathtt{p} \equiv \mathtt{q} \ \vee \ & \exists \mathtt{s}.\exists \mathtt{r},\mathtt{t}.(\mathtt{s} \not\equiv \varepsilon \wedge (\exists \mathcal{S} \in \Pi.\mathtt{r} \in \mathcal{S} \wedge \mathtt{t} \in \mathcal{S}) \\ & \wedge \mathtt{p} \equiv (\mathtt{p}' \cdot \mathtt{s}) \wedge \mathtt{q} \equiv (\mathtt{q}' \cdot \mathtt{s}) \wedge \mathsf{alias}(\mathtt{p}',\mathtt{r}) \wedge \mathsf{alias}(\mathtt{q}',\mathtt{t}))\end{aligned}$$

in which '$\equiv$' and '$\not\equiv$' are relations denoting equality and non-equality in syntax respectively, ε is empty string and '$\cdot$' is the string appending operation. If `p` and `q` are syntactically equal then they are aliases. And if they are not, then judge the two access paths $\mathtt{p}'$ and $\mathtt{q}'$ by removing the same suffix `s`. If the alias `r` of $\mathtt{p}'$ and the alias `t` of $\mathtt{q}'$ appear in the same subset of Π, then `p` and `q` are aliases. If this deduce depends on the alias relation of p and q itself, the deduce is failed (they are not considered to be aliases).

Legal Ψ should satisfy the following conditions:

(1) All declared pointer variables must appear in Ψ.

(2) For each pointer `p` of Π, if it points to a structure with a pointer field named `r`, then some alias of `p->r` is in Ψ.

(3) Any two different pointers in Ψ are not aliases.

(4) Every prefix of each pointer in Ψ has an alias in Π.

If one pointer is an alias of some pointer in a legal Ψ, then it is called *legal pointer*. Integer typed access path is legal, if its prefixes are all legal pointers. We use $\mathcal{P}$ to stand for all the legal access paths at a program point.

One access path is called *access path with circle*, if aliases of two different prefixes of it appear in the same subset of Π. Access path without circles are called *minimal access path*. We assume that there are only minimal access paths in programs.

3 Design of Pointer Logic

To prove that programs satisfy the basic safety policy, we have designed a logic system for PointerC language besides a type system. There are side conditions in some typing rules. For example, access paths in program expressions must be legal and function call to `free` should not cause memory leaks. Since these side conditions cannot be checked by traditional type systems, we use the pointer logic to prove them. In this section, we introduce the redesigned pointer logic which is based on the concept and rules of legal Ψ, while the earlier version is based on functions calculating the alias sets of pointers. Next the assertion language will be expatiated first.

3.1 Assertion Language

The syntax of the assertion language is shown in Figure 3.

$$
\begin{aligned}
\mathit{assertion} ::= {} & \mathit{boolexp} \mid \neg\ \mathit{assertion} \mid \mathit{assertion}\ \wedge\ \mathit{assertion} \mid \mathit{assertion}\ \vee\ \mathit{assertion} \\
\mid {} & \mathit{assertion}\ \Rightarrow \mathit{assertion} \mid \mathit{ident}(\mathit{lval}) \mid (\mathit{assertion}) \\
\mid {} & \exists \mathit{ident} : \mathit{domain}.\mathit{assertion} \mid \forall \mathit{ident} : \mathit{domain}.\mathit{assertion} \\
\mid {} & \{\mathit{lvalset}\} \mid \{\mathit{lvalset}\}_{\mathcal{N}} \mid \{\mathit{lvalset}\}_{\mathcal{D}} \\
\mathit{domain} ::= {} & \mathbf{N} \mid \mathit{exp}..\mathit{exp} \quad \mathit{lvalset} ::= \mathit{lvalset},\ \mathit{lval} \mid \mathit{lval} \quad \mathit{lval} ::= \cdots \mid \mathit{lval}(\texttt{->}\mathit{ident})^{\mathit{exp}}
\end{aligned}
$$

Fig. 3. Assertion Language of the Pointer Logic

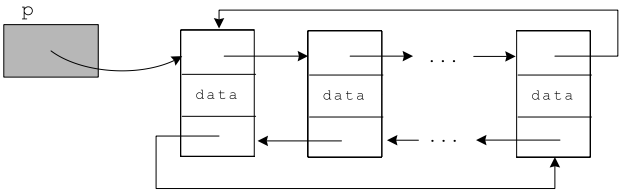


Fig. 4. Circular Doubly-linked List

In Figure 3, the syntax of *exp*, *boolexp* and *lval* refer to the syntax of PointerC language (Figure 1). $\{lvalset\}$ is used to represent access path set of valid pointers ($\mathcal{S}$ is used to stand for such a set). $\{lvalset\}_{\mathcal{N}}$ and $\{lvalset\}_{\mathcal{D}}$ are used to stand for access path set of `NULL` pointers and dangling pointers respectively. If $\mathcal{S}_1$,..., $\mathcal{S}_n$ are access path sets of valid pointers at a program point, then $\Pi = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ or $\Pi = \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n$ (two different shorthands). $lval(\texttt{->}ident)^{exp}$ is a new form of access path for assertion only (named access path with superscript). For example, $\texttt{s(->next)}^i$ is shorthand for $\texttt{s->next->next}\cdots\texttt{->next}$(in which `->next` appears i times). If i equals to zero, then $\texttt{s(->next)}^i$ is `s` itself.

Based on this grammar of assertion language, we give more restrictions on assertions:

(1) **N** stands for natural number. And expressions in *exp*..*exp* and $lval(\texttt{->}ident)^{exp}$ are restricted to integer expressions `n`$\pm$`b` or `b` where `n` is a bound variable and `b` can be linear addition/subtration expression including constant and at most one program variable.

(2) When a set of access paths is bounded by several quantifiers, the existential quantifier must be the outermost quantifier if the existential quantifier exists.

(3) Logic negation operator ($\neg$) can not apply to sets of access paths.

For example,$\forall$`i:0..n-1.`$\{\texttt{p->l(->r)}^i\}$ is shorthand for

$$\{\texttt{p->l}\}\wedge\{\texttt{p->l->r}\}\wedge\ldots\wedge\{\texttt{p->l(->r)}^{n-1}\}.$$

These sets of access paths are sets of Π in Ψ. Thanks to quantifiers and access paths with superscript, circular doubly-linked list in Figure 4 can be defined using our assertion language as follows:

$$\texttt{dlist(p,n)}\triangleq(\forall\texttt{k:1..n.}\{\texttt{p(->r)}^k,\texttt{p(->r)}^{k+1}\texttt{->l}\})\wedge\{\texttt{p,p(->r)}^{n+1}\texttt{, p->r->l}\},$$

in which `p` is a pointer pointing to structure `node` (mentioned in section 2) and n (positive integer) is the number of nodes in the list except the head node. The last set bounded by universal quantifier k is $\{\texttt{p(->r)}^k\texttt{, p(->r)}^{k+1}\texttt{->l}\}$ where $k = n$. It can also be expressed as $\{\texttt{p(->r)}^n\texttt{, p->l}\}$ so as to delete the circle in the access path $\texttt{p(->r)}^{n+1}\texttt{->l}$. We use the first form in order to make the definition uniform in denoting all the nodes except the head one. So we allow access paths with circle to appear in assertions. The definition of circular doubly-linked list gives a clear equality relation among the $2n + 3$ pointers in Figure 4. And for every pointer in Figure 4, this definition only chooses one access path from its alias set.

Definition of some data structure in our assertion language needs to be given by

induction. For example, the predicate for binary tree is as follows:

$$\texttt{tree(p)} \triangleq \{\texttt{p}\}_{\mathcal{N}} \vee (\{\texttt{p}\} \wedge \texttt{tree(p->l)} \wedge \texttt{tree(p->r)}),$$

in which p is still a pointer pointing to structure node. $\{\texttt{p}\}_{\mathcal{N}}$ is denote for empty tree and $\{\texttt{p}\} \wedge \texttt{tree(p->l)} \wedge \texttt{tree(p->r)}$ is used to denote non-empty tree. By using valid pointer set, it is convenient to express all of the valid pointers in the tree are not equal to each other. Because when expanding all references of the inductive definitions, all of valid pointers appear in different subset of Π.

3.2 Assertion Calculus

Using equivalence axioms of conjunction and disjunction, assertions including access path set can be transformed into disjunctive normal form (DNF). During the transformation, each access path set is regarded as a logic constant. In the DNF assertion, access path sets of each clause form a Ψ. It is remarkable that there could be other kinds of assertion, some of which are even related to judging the legalness of Ψ. For example, predicates using inductive definitions and assertions about integer variables in superscript expressions of access paths should appear in the clause of DNF.

According to the principle that no pointer access path information should be dropped, the classical axiom of implication $A \wedge B \Longrightarrow A$ can not be used in pointer logic reasoning when B includes access path sets or use of inductive definitions except that A is the same with B. Because it results in some pointer information lost. And the axiom $A \wedge B \Longrightarrow B$ is in a similar case.

Moreover, we need to give some equivalence and implication axioms for Ψ.

1) **Equivalence axioms of $\mathcal{N}$and $\mathcal{D}$**

$$\begin{array}{ll} \mathcal{N}_1 \wedge \mathcal{N}_2 \Leftrightarrow \mathcal{N} & (\texttt{if}(\mathcal{N} == \mathcal{N}_1 \cup \mathcal{N}_2) \wedge (\mathcal{N}_1 \cap \mathcal{N}_2 == \emptyset)) \\ \mathcal{D}_1 \wedge \mathcal{D}_2 \Leftrightarrow \mathcal{D} & (\texttt{if}(\mathcal{D} == \mathcal{D}_1 \cup \mathcal{D}_2) \wedge (\mathcal{D}_1 \cap \mathcal{D}_2 == \emptyset)) \end{array}$$

   The equivalence axiom of $\mathcal{D}$ is similar. These two equivalences depend on whether the pointer information included by the two sides are equal.

2) **Axioms of illegal Ψ and assertion**
   Previous mentioned conditions for legal Ψ to satisfy should be supplemented after quantifier and inductive definition are introduced into the assertion language:
   (1) Quantifiers don't change the conditions for legal Ψ to satisfy. But it is worth noting that quantifiers bring more difficulties to the alias reasoning between the pointers because equality reasoning between integer superscript expressions involves.
   (2) Inductive definition body should be checked. The body is a DNF. All the Ψ formed from clauses of the body should be legal.
   (3) Arguments of every reference of inductive definitions should be considered. Since the inductive definitions is checked, one solution is adding them into NULL pointer set $\mathcal{N}$ before the check instead of expanding every reference using the body of inductive definitions. It is feasible because adding the

arguments of every reference of inductive definitions into $\mathcal{N}$ means we don't care about the unexpanded part of the definitions temporarily.

(4) These conditions should be satisfied by all the Ψ formed from clauses of a DNF assertion.

Now we can extend the legal Ψ to legal assertion. A clause of DNF assertion can be transformed into $\Psi \wedge Q$ where Q is assertion not including access path sets. If Ψ is legal and every access path in Q is legal, then the clause (that is $\Psi \wedge Q$) is legal. If all clauses of DNF assertion are legal, then it is a legal assertion. The non-legal assertion axioms are:

$$\Psi \Longrightarrow false \qquad \text{(if } \Psi \text{ is not legal)}$$
$$\Psi \wedge Q \Longrightarrow false \qquad \text{(if } \Psi \text{ or } Q \text{ is not legal)}$$

3) **Equivalence axioms of access path set and assertion**

(1) If Ψ is legal, access path set $\mathcal{R}_1$ and $\mathcal{R}_2$ are two sets in Ψ, and the following proposition holds:

$$|\mathcal{R}_1| == |\mathcal{R}_2| \wedge \forall \mathtt{p}{:}\mathcal{R}_1.\exists \mathtt{q}{:}\mathcal{R}_2.\mathsf{alias}(\mathtt{p,\ q}) \wedge \forall \mathtt{q}{:}\mathcal{R}_2.\exists \mathtt{p}{:}\mathcal{R}_1.\mathsf{alias}(\mathtt{q,\ p}),$$

then $\mathcal{R}_1$ and $\mathcal{R}_2$ are equivalent ($|\mathcal{R}|$ is the size of $\mathcal{R}$). So the equivalence axiom of access path set is written as:

$$\mathcal{R}_1 \Leftrightarrow \mathcal{R}_2 \quad \text{(If } \Psi \text{ is legal and } \mathcal{R}_1, \mathcal{R}_2 \text{ are equivalent)}$$

(2) Equivalence axiom of Ψ is as follows:

$$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q \Leftrightarrow \mathcal{S}'_1 \wedge \ldots \wedge \mathcal{S}'_n \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q$$

(If the right and left Ψ and Q are all legal, and the corresponding set are equivalent based on the same Ψ of the two sides)

$$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \wedge \mathcal{N} \wedge \mathcal{D} \wedge (\mathtt{p==NULL}) \wedge Q \Longrightarrow false$$
$$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N}^p \wedge \mathcal{D} \wedge (\mathtt{p==NULL}) \wedge Q \Longrightarrow \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N}^p \wedge \mathcal{D} \wedge Q$$
$$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p,q} \wedge \mathcal{N} \wedge \mathcal{D} \wedge (\mathtt{p!=q}) \wedge Q \Longrightarrow false$$
$$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \wedge \mathcal{N} \wedge \mathcal{D} \wedge (\mathtt{p!=NULL}) \wedge Q \Longrightarrow \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^p \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q$$
$$\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p,q} \wedge \mathcal{N} \wedge \mathcal{D} \wedge (\mathtt{p==q}) \wedge Q \Longrightarrow \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_{n-1} \wedge \mathcal{S}^{p,q} \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q$$

Fig. 5. Axiom Schemas of Boolean Expressions and Ψ

4) **Axiom schemas of boolean expressions and Ψ**

Boolean expressions such as `p==NULL, p!=NULL, p==q and p!=q` (`p` and `q` are all pointers) will be added to assertion when reasoning the programs by *conditional rule* or *while rule* of Hoare logic. The representative axiom schemas in Figure 5 are designed to apply to these cases. If the pointer information is consistent with Ψ then it is absorbed. Otherwise, it will lead to inconsistent pointer information (that is false). $\mathcal{S}^p$ is short for a valid pointer set which includes an alias of pointer `p`. Similarly, $\mathcal{S}^{p,q}$ is short for a valid pointer set which includes an alias of pointer `p` and an alias of `q`.

For example, under the definition of the binary tree mentioned before, then:

$$
\begin{aligned}
&\texttt{tree(p)}\wedge\texttt{p!=NULL}\\
\equiv\ &(\{\texttt{p}\}_{\mathcal{N}}\vee(\{\texttt{p}\}\wedge\texttt{tree(p->l)}\wedge\texttt{tree(p->r)}))\wedge\texttt{p!=NULL}\\
\Longrightarrow\ &\{\texttt{p}\}_{\mathcal{N}}\wedge\texttt{p!=NULL}\ \vee\ \{\texttt{p}\}\wedge\texttt{tree(p->l)}\wedge\texttt{tree(p->r)}\wedge\texttt{p!=NULL}\\
\Longrightarrow\ &\texttt{false}\ \vee\ \{\texttt{p}\}\wedge\texttt{tree(p->l)}\wedge\texttt{tree(p->r)}\\
\Longrightarrow\ &\{\texttt{p}\}\wedge\texttt{tree(p->l)}\wedge\texttt{tree(p->r)}
\end{aligned}
$$

3.3 Inference rules of pointer logic

In this subsection, we will introduce the inference rules in detail, but firstly we give some definitions of the basic operations on access path set and predicates. They are used to figure out Ψ at each program point.

(1) **Access path deletion.** This operation gives the result set by deleting an alias or aliases of given pointer(s) from the access path set $\mathcal{R}$.

$$\mathcal{R}-\texttt{p}\triangleq\{\texttt{q}\ :\ \mathcal{R}\mid\neg\mathsf{alias}(\texttt{q},\ \texttt{p})\}$$
$$\mathcal{R}-\{\texttt{p}_1,\ \ldots,\ \texttt{p}_n\}\triangleq((\mathcal{R}-\texttt{p}_1)-\ldots-\texttt{p}_n)$$

(2) **Access path addition.** This operation returns the result set by adding given pointer(s) to access path set $\mathcal{R}$.

$$\mathcal{R}+\texttt{p}\triangleq\mathcal{R}\cup\{\texttt{p}\}\quad\mathcal{R}+\mathcal{R}'\triangleq\mathcal{R}\cup\mathcal{R}'$$

(3) **Prefix substitution.** For each access path q in the given set $\mathcal{R}$, if a prefix of q is an alias of p, q is substituted by its alias $\texttt{q}'$ where $\texttt{q}'$ is not prefixed by any alias of p. Other access paths are not changed. Predicate prefix(r, q) is used to describe that r is a prefix of q.

$$\mathcal{R}/\texttt{p}\ \triangleq\ \mathcal{R}'\ \texttt{where}\ \mathcal{R}'\Leftrightarrow\mathcal{R}\wedge\forall\texttt{q}{:}\mathcal{R}'.(\neg\exists\texttt{r}{:}\mathcal{P}.(\mathsf{alias}(\texttt{r},\texttt{p})\wedge\mathsf{prefix}(\texttt{r},\texttt{q})))$$

Similarly, $Q/\texttt{p}$ is defined.

(4) **Predicate leak($\mathcal{S}$, p).** When deleting access path p from a valid pointer set $\mathcal{S}$ of Π (possibly when p is assigned), if all access paths in $\mathcal{S}$ are aliases of p or are prefixed by p, then there exist memory leaks.

$$\mathsf{leak}(\mathcal{S},\texttt{p})\triangleq\forall\texttt{q}{:}\mathcal{S}.(\mathsf{alias}(\texttt{q},\texttt{p})\vee\exists\texttt{r}{:}\mathcal{P}.(\mathsf{alias}(\texttt{r},\texttt{p})\wedge\mathsf{prefix}(\texttt{r},\texttt{q})))$$

Next the axioms and inference rules for statements of PointerC are presented. In these rules, effects on the program state by each statement are reflected in deletions, additions and substitutions on these access path sets. In the following rules, reasoning on access paths are based on the Ψ at the program point before the statement is executed. And all the boolean expressions which can be absorbed are all absorbed using rules given in Section 3.2.

1) **Assignment statement of pointers (p = q and p = NULL)**
Due to the limited space, one typical case is given as follows and rules for other cases can be figured out similarly. When an alias of p is in some valid access path set of Π and alias of q is in $\mathcal{N}$, the rule is:
$\{\!\{\mathcal{S}_1\wedge\ldots\wedge\mathcal{S}_{n-1}\wedge\mathcal{S}^p\wedge\mathcal{N}^q\wedge\mathcal{D}\wedge Q\}\!\}$
p = q

$\{\!\{\mathcal{S}_1/\mathtt{p} \wedge \ldots \wedge \mathcal{S}_{n-1}/\mathtt{p} \wedge (\mathcal{S}^p/\mathtt{p}-\mathtt{p}) \wedge (\mathcal{N}^q/\mathtt{p}+\mathtt{p}) \wedge \mathcal{D}/\mathtt{p} \wedge Q/\mathtt{p}\}\!\}$
(if $\mathsf{leak}(\mathcal{S}^p,\mathtt{p})$ is false based on the Ψ formed by the precondition)

2) **Assignment statement of non-pointers (`x = e`)**

$\{\!\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D} \wedge Q\}\!\}$
`x = e`
$\{\!\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D} \wedge (\exists \mathrm{x}'.(\mathrm{x}==\mathrm{e}[\mathrm{x}\leftarrow\mathrm{x}'] \wedge Q[\mathrm{y}_1\leftarrow\mathrm{x}]\ldots[\mathrm{y}_n\leftarrow\mathrm{x}][\mathrm{x}\leftarrow\mathrm{x}'])\}\!\}$
($\mathrm{y}_1,\ldots,\mathrm{y}_n$ are all aliases of x presented in Q,$\mathrm{x}' \notin (\{\mathrm{x}\} \cup \mathrm{FV}(\mathrm{e}) \cup \mathrm{FV}(Q))$),

in which FV maps expressions and assertions to the set of free variables in them.

3) **Composition/Conditional/While rule**
These rules and the *consequence rule* are the same to those in Hoare logic.

4) **Malloc rule**
This rule is for statement `p = malloc(struct t)` where `t` is a structure type and $\mathtt{r}_1,\ldots,\mathtt{r}_n$ are pointer typed field names of structure `t`. One case is given, and other cases are similar. If an alias of `p` is in $\mathcal{N}$, the rule is:
$\{\!\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N}^p \wedge \mathcal{D} \wedge Q\}\!\}$
`p = malloc(struct t)`
$\{\!\{\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \{\mathtt{p}\} \wedge (\mathcal{N}^p-\mathtt{p}) \wedge (\mathcal{D}+\{\mathtt{p->r}_1,\ldots,\mathtt{p->r}_n\}) \wedge Q\}\!\}$

5) **Case analysis rule**
If the assertion is a DNF assertion, then this rule can be applied first to do case analysis.

$$\frac{\{\!\{\Psi_1 \wedge Q_1\}\!\}\mathtt{C}\{\!\{\Psi_3 \wedge Q_3\}\!\} \qquad \{\!\{\Psi_2 \wedge Q_2\}\!\}\mathtt{C}\{\!\{\Psi_4 \wedge Q_4\}\!\}}{\{\!\{\Psi_1 \wedge Q_1 \vee \Psi_2 \wedge Q_2\}\!\}\mathtt{C}\{\!\{\Psi_3 \wedge Q_3 \vee \Psi_4 \wedge Q_4\}\!\}}$$

where `C` is a program segment. Although the precondition `x == 1` $\vee$ `x == 2` can be regarded as two cases, there is no such rule in Hoare logic because assignment axiom can do substitution simultaneously to each clause of the DNF assertion.

6) **Frame Rule**
In order to obtain more effective and modular reasoning method, sometimes Ψ must be splitted or combined. If a legal Ψ (according to declared pointer variables in set Δ) can be written as $\mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N} \wedge \mathcal{D}$,

$$\Psi_1 = \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_i \wedge \mathcal{N}_1 \wedge \mathcal{D}_1 \qquad \Psi_2 = \mathcal{S}_{i+1} \wedge \ldots \wedge \mathcal{S}_n \wedge \mathcal{N}_2 \wedge \mathcal{D}_2$$

where $\mathcal{N}_1 \cup \mathcal{N}_2 \Leftrightarrow \mathcal{N}$ and $\mathcal{D}_1 \cup \mathcal{D}_2 \Leftrightarrow \mathcal{D}$. If Ψ_1 and Ψ_2 are legal according to declared pointer variables in set Δ_1, Δ_2 respectively ($\Delta_1 \cup \Delta_2 = \Delta$ and $\Delta_1 \cap \Delta_2 = \emptyset$), then Ψ can be splitted as legal Ψ_1 and Ψ_2; Ψ_1 and Ψ_2 can be combined as Ψ. Obviously here $\Psi \Leftrightarrow \Psi_1 \wedge \Psi_2$ holds. Q can be splitted and combined in a similar way. The rule for local reasoning is:

$$\frac{\{\!\{\Psi_1 \wedge Q_1\}\!\}\mathsf{C}\{\!\{\Psi_1' \wedge Q_1'\}\!\}}{\{\!\{(\Psi_1 \wedge \Psi_2) \wedge (Q_1 \wedge Q_2)\}\!\}\mathsf{C}\{\!\{(\Psi_1' \wedge \Psi_2) \wedge (Q_1' \wedge Q_2)\}\!\}}$$

(where both $\Psi_1 \wedge Q_1$ and $\Psi_2 \wedge Q_2$ are legal)
For example, at the program point of function calls, $\Psi_1 \wedge Q_1$ can be used to represent assertions concerned with the called function and $\Psi_2 \wedge Q_2$ is not concerned with the called function.

3.4 Inference rule for function construct

Besides the syntax features introduced in Section 2, PointerC also supports function definitions and function calls. This subsection discusses inference rules for the function construct. To simplify the discussion, the restrictions or assumptions are listed as follows.

(1) The form of function definition is

$$\texttt{f(arg)\{vardeclist stmtlist\}},$$

where `arg` is the only parameter of `f` ,`vardeclist` is the local variables and `stmtlist` is list of statements. And function call can only be used in an assignment statement like `ret = f(act)`, where `act` is the actual parameter. In the following rules, $\texttt{v}_1$,..., $\texttt{v}_k$ are all declared local pointer variables of function `f`.

(2) Local variable `res` is a special virtual variable for reasoning the `return` statement. It is initially a dangling pointer.The reasoning of `return e` can be done by reasoning the sequence `res = e; return res` in order to simplify the reasoning.

To reason the legalness of Ψ, declared pointer variables set is given. To support function construct, it is based on the static scope rule to select which pointer variable should be included in Ψ.

The following rules are for the case that both the parameter and return value are pointer type and the other cases are much easier. And rules for recursive function, procedure and recursive procedure are easy to figure out based on these rules.

1) **Function definition.**

$$\frac{\{\!\{(\Psi \wedge \{\texttt{v}_1, \ldots, \texttt{v}_k, \texttt{res}\}_{\mathcal{D}}) \wedge Q\}\!\}\texttt{stmlist}\{\!\{\Psi' \wedge Q'\}\!\}}{\{\!\{\Psi \wedge Q\}\!\}\texttt{f(arg)\{vardeclist stmlist\}}\{\!\{\Psi' \wedge Q'\}\!\}}$$

To certify the fuction body `stmlist`, $\{\texttt{v}_1,\ldots, \texttt{v}_k, \texttt{res}\}_{\mathcal{D}}$ is added into the function precondition because local pointer variables and `res` are regarded as dangling pointers before the body is executed.

2) **Function call.**

$$\frac{\{\!\{\Psi \wedge Q\}\!\}\texttt{f(arg)\{vardeclist stmlist\}}\{\!\{\Psi' \wedge Q'\}\!\}}{\{\!\{(\Psi \wedge \{\texttt{ret}\}_{\mathcal{N}} \wedge Q)[\texttt{arg} \leftarrow \texttt{act}]\}\!\}\texttt{ret = f(act)}\{\!\{(\Psi' \wedge Q')[\texttt{res} \leftarrow \texttt{ret}]\}\!\}}$$

If $\Psi \wedge Q$ is only concerned with `arg`, `act` and `ret` must be declared variables because if they are access paths like `s->next`, then $\{\Psi \wedge \{\texttt{ret}\}_{\mathcal{N}} \wedge Q\}$ [`arg`←`act`] can not be legal. In order to simplify the rule, `ret` must be a `NULL` pointer before function call.

3) **Return statement.**

$$\frac{\{\!\{\Psi_0 \wedge Q_0\}\!\}\mathtt{v}_1 \ = \ \mathtt{NULL}\{\!\{\Psi_1 \wedge Q_1\}\!\} \ldots \quad \{\!\{\Psi_{k-1} \wedge Q_{k-1}\}\!\}\mathtt{v}_k \ = \ \mathtt{NULL}\{\!\{\Psi_k \wedge Q_k\}\!\} \quad \{\!\{\Psi_k \wedge Q_k\}\!\}\mathtt{arg} \ = \ \mathtt{NULL}\{\!\{\Psi_{k+1} \wedge Q_{k+1}\}\!\}}{\{\!\{\Psi_0 \wedge Q_0\}\!\}\mathtt{return\ res}\{\!\{\Pi_{k+1} \wedge (\mathcal{N}_{k+1} - \{\mathtt{v}_1, \ldots, \mathtt{v}_k, \mathtt{arg}\}) \wedge \mathcal{D}_{k+1} \wedge Q_{k+1}\}\!\}}$$

After return statement, the declared local pointer variables $\mathtt{v}_1$,..., $\mathtt{v}_k$ and parameter `arg` can not be accessed, they must be removed from the assertions.

4 Implementation of Pointer Logic in Coq

We have completed the safety theorem proof of the PointerC language and the soundness theorem proof of the former version pointer logic [10] using the proof assistant Coq [11]. The soundness theorem ensures that the axioms and rules of pointer logic are sound with respect to the operational semantics of the PointerC language. We have implemented a core part of redesigned pointer logic and verified three non-trivial examples in the proof assistant Coq [18]. The implementation mainly includes:

(1) **PointerC language**. Syntax, types and typing rules of the PointerC language are formalized.

(2) **Library**. Lemmas of set operations and other utilities.

(3) **Pointer logic**. First, we formalized the syntax of assertion language based on syntax of PointerC. Then rules for legal assertions and legal Ψ are given. Finally, we implemented inference rules and axioms of the pointer logic. Since our redesign mainly adds restrictions on the assertions, the rules are almost the same. The soundness of the redesigned pointer logic can be similarly proved.

(4) **Examples**. We have verified three examples to show the usability of the pointer logic. The programs are operations on singly-linked list, binary tree and circular doubly-linked list. For each function, user must provide the pre-condition, post-condition and loop invariant for each loop.

5 Case study

Using a certifying compiler prototype for the PointerC language, we have proved the safety properties or partial correctness of some simple functions concerning shared mutable data structures such as singly linked list, circular doubly-linked list and binary tree.

We take the function

```
struct node *DeleteNode(struct node *p)
```

as the second example to show the application of the pointer logic, using the definition of tree node in section 2.

The function deletes a node from a binary search tree pointed by the parameter `p` and reconnects its left or right child. The precise layout of valid pointers and null pointers in the tree pointed by `p` is unknown, but the tree pointed by `p` must meet the definition `tree(p)` which is defined in section 3.

The annotated code of the function `DeleteNode` is shown in Figure 6. The assertions are inserted in terms of the pointer logic. For the conditional branch which states that neither the left nor the right child of the parameter `p` is null (the function requires `p` is not null), assertions are inserted at most of the program points; for other parts of the code, assertions are inserted only at some key points.

6 Comparison with separation logic

Pointer logic and separation logic [15,16] are both extensions of Hoare logic that permit reasoning about imperative programs with shared mutable data structures. Although it has some inference rules for high-level control structures (such as conditional and while rules), separation logic is still not suitable to reason about high-level imperative programs using long access paths directly. In separation logic, assertions P and Q in separation conjunction $P * Q$ hold for disjoint portions of the addressable storage. And then in the programming languages related to separation logic, a sequence of dereference operations is not allowed to appear in one expression. But such operations are often used in high-level imperative programs. For example, the following program segment written in C appears in some function of deleting an element from a singly-linked list:

```
s = t->next;t->next = t->next->next; free(s);
```

Expression `t->next->next` is related to two separated heap blocks. There is no inference rule in separation logic which can be applied to such expressions. The following code which is transformed from the above code:

```
s = t->next; r = s->next; t->next = r; free(s);
```

can be verified in separation logic. But such transformation is not always trivial because alias analysis is needed in some cases.

While extending Hoare logic, the strategy of separation logic can be understood as follows: all pointers are assumed to potentially point to the same object, unless they are explicitly expressed to point to different objects. Therefore, separation logic needs to introduce new connectives such as separation conjunction ($*$). The disadvantage of this method is mentioned above. We believe that the most important thing in reasoning pointer programs is to hold the equality information among valid pointers at each program point. Based on the pointer information, we can infer whether or not two access paths are aliases for each other, and then overcome the difficulty that aliasing brings to Hoare logic. The start point of our extension of Hoare logic is that different access paths are assumed to represent different pointers, unless it can be proved from the collected pointer information that they are aliases for each other. The inference rules of the pointer logic are used to deduce the pointer information at the program point after a statement from the pointer

```
{{{p}∧tree(p)}}
struct node * DeleteNode (struct node *p)
{ struct node *q,*s;
    {{{p}∧{q,s,res}_D∧tree(p)}}
  if(p->r==NULL) /* right child is null, reconnect left child */
    {q = p; s = p->l; free(q); {{{p,q,res}_D∧tree(s)}} return s;}
  else if(p->l==NULL) /* left child is null, reconnect right child
*/
        { q = p; s = p->r; free(q);{{{s}∧{p,q,res}_D∧tree(s)}} return
s;}
  else /* neither left nor right child is null */
    { {{{p}∧{p->l}∧{p->r}∧{q,s,res}_D∧tree(p->l)∧tree(p->r)}}
    q = p; s = p->l;
    if(s->r == NULL) /* reconnect *q's left child */
      { q->l = s->l; p->data = s->data; free(s);
{{{p,q}∧{s,res}_D∧tree(p)}} return p; }
    else
    {   {{{p,q}∧{p->r}∧{p->l,s}∧{s->r}∧{res}_D∧tree(p->l->l)∧tree(p->l->r)∧tree(p->r)}}
      q = s; s = s->r;
        {{{∃n:N.({p}∧{p->r}∧∀i:0..n-1.{p->l(->r)^i}∧{p->l(->r)^n,q}∧{p->l(->r)^{n+1},s}
          ∧{res}_D∧∀i:0..n.tree(p->l(->r)^i->l)∧tree(p->l(->r)^{n+1})∧tree(p->r))}}}
      /* loop invariant */
      while(s->r != NULL) /* turn left, then go on end of the right
side */
      { q = s; s = s->r;}
        {{{∃n:N.({p}∧{p->r}∧∀i:0..n-1.{p->l(->r)^i}∧{p->l(->r)^n,q}∧{p->l(->r)^{n+1},s}
          ∧{s->r}_N∧{res}_D∧∀i:0..n.tree(p->l(->r)^i->l)∧tree(p->l(->r)^{n+1})∧tree(p->r))}}}
      p->data = s->data;
      q->r = s->l; /* reconnect *q's right child */
        {{{∃n:N.({p}∧{p->r}∧∀i:0..n-1.{p->l(->r)^i}∧{p->l(->r)^n,q}∧{s}
          ∧(({s->l,q->r}∧{s->r}_N)∨{s->r,s->l,q->r}_N)∧{res}_D)
            ∧∀i:0..n.tree(p->l(->r)^i->l)∧tree(p->l(->r)^{n+1})∧tree(p->r))}}}
      free(s);
        {{{∃n:N.({p}∧{p->r}∧∀i:0..n-1.{p->l(->r)^i}∧{p->l(->r)^n,q}∧({q->r}∨{q->r}_N)
          ∧{s,res}_D∧∀i:0..n.tree(p->l(->r)^i->l)∧tree(p->l(->r)^{n+1})∧tree(p->r))}}}
      return p;
        {{{res}∧tree(res)}}
     }
   }
}
```

Fig. 6. Example of Deleting a Node from a Binary Search Tree

information at the program point before the statement. The information can also be used to deduce other properties of programs.

Further more, separation logic does not care memory leaks, and its inference rules do not rule out the commands that may cause memory leaks, although programmer can do it using specific post-conditions for programs. On the contrary, the pointer

logic refuses any command that causes memory leaks. Usually, a memory leak means that there exists heap blocks which can not be traced from store variables. For example, the following program segment with assertions:

$$\{\!\{\texttt{emp}\}\!\}\ \texttt{x=cons(10)}\ \{\!\{\texttt{x}\mapsto\texttt{10}\}\!\}\ \texttt{x=nil}\ \{\!\{\exists \texttt{x}.\texttt{x}\mapsto\texttt{10}\}\!\};$$

can be verified using separation logic, but there is a memory leak when `x` is assigned `nil`. It can not be ruled out because using assignment axiom of Hoare logic for the statement `x=nil`. In order to solve this problem, Reynolds *et al.* presented the concept of precise assertions [16], but its definition is based on the semantics of abstract machine instead of the assertion syntax. The details for the relations between pointer logic and separation logic is provided in [14].

If two restrictions as follows are given on the syntax of PointerC and the assertion, the pointer logic is another form of separation logic in essence. And under these two restrictions, alias reasoning can be simplified to a large extent.

(1) `lval` is restricted as

```
lval ::= ident | ident->ident,
```

and `ident->ident` can only appear in one side of the assignment statement.

(2) There is no superscript expression in `lval` of the assertion language.

This form of the pointer logic is different from separation logic in the following two points. (1)There are several high-level languages features, such as types, access via field name not offset. (2)Using access path set in assertions, there is no need to use separation connectives but can express more meanings.

And the strongpoints lie in: the pointer logic can be used to verify programs written in C-like high level languages without adding restrictions on access paths; the calculus of assertion is simple and clear in contrast to separation logic; there will be no memory leaks in the verified programs by the pointer logic.

7 Related Work

An important characteristic of Hoare logic is its use of variable substitution to capture the semantics of assignments. Pointer logic essentially uses the idea of precise alias analysis in program verification. But one of the prominent advantages is using access path sets to represent pointer information of program points. Another advantage is to express the effects of statements to pointer information using Hoare-style inference rules. So pointer logic can be used in Hoare-style verification of high-level programs.

Pointer information can be represented by many methods such as point-to graph, point-to set and so on. Jonkers pioneered in introducing the storeless model [19] to abstract memory address and use access path sets to represent pointer information. In the storeless model, each dynamically allocated object is denoted by a set of access paths which can reach it from declared variables. And thus the heap can be represented by a group of such sets. Alias of an access path also appears in the same set and it makes the information redundant. Recent researches using storeless model [20,21] still use such representations. In our pointer logic, aliases of the access path are not included in the set, but they can be inferred if needed. So data

structures with cycles such as circular doubly-linked list can be better represented in our method.

C.A.R. Hoare and Jifeng He's *Trace Model* supports assertions about the general topology of pointer based data structures [22]. The model deals with aliasing directly by recording paths of field names (namely *traces*) which denote the same object. We also use access paths similar to traces in essence, but we don't record all the traces because some traces can be inferred by other traces. For example, in Figure 2, traces for the object O_2 are $root \xrightarrow{head} O_1 \xrightarrow{next} O_2$ (that is, access path `head->next`) and $root \xrightarrow{p} O_1 \xrightarrow{next} O_2$ (access `path p->next`). In our method, only one of the two trace is selected (for details, see section 2). Furthermore, we classify the access paths by what object they points to. With the help of Π, $\mathcal{N}$ and $\mathcal{D}$, it is easy to maintain precise pointer information at any program point concisely.

Common pointer analysis uses program analysis to obtain an estimate of the pointer information without any given information. And we reason about the precise pointer information at each program points under the help of pre- and post-condition of functions and loop invariants. Pointer analysis has been studied for more than 20 years, and in history it mainly tried to answer the question: what is the possible set of objects pointed by a pointer at runtime? Such pointer analysis can be used in many fields of static analysis of programs and program optimization: liveness analysis needed by register allocation and constant propagation, and static checking of runtime errors such as dereference of null pointers. In recent years, it has also been used to discover harmful buffer overflows and format string attacks. Similarly to other static techniques, pointer analysis is bothered by the problem of decidability, so that for most languages, the solution is always an approximation.

For the requirement of software safety, our pointer logic needs to realize an accurate instead of approximate pointer analysis. Therefore, on the premise of not affecting the functionality of the language, we have restricted some undecidable operations of pointers in PointerC. And it is the restriction which makes it possible to express the collection of pointer information using the inference rules of the pointer logic.

As for proving program properties, Bornat also used Hoare logic to reason about properties of pointer programs [23]. Inspired by the work of Burstall [24] and Morris [25], he treated memory heap as a pointer-indexed collection of objects, each of which is a name-indexed collection of components. An object-component reference in the heap corresponds to a double indexing, one of the heap and one of the object. He introduced axioms for object component substitution for distinct component names and used them to prove properties of programs with shared mutable data structures defined by pointers. Mehta and Nipkow adopted a similar approach when reasoning about pointer programs in a higher-order logic system Isabelle/HOL [26]. And so did Marché *et al.* in their prototype tool Caduceus [27]. Both approaches of Bornat's and ours can be considered to solve aliasing by using the double indexing. The axioms and rules introduced by Bornat are concise and used in weakest precondition calculus, our axioms and rules are not as concise as Bornat's ones but collect more pointer information in a forward manner. Our approach has the following advantages comparing with Bornat's approach. PointerC provides explicit deallocation operation and the pointer logic can be used to detect memory leak,

but Bornat's approach can only be used in the situation where heap management depends on garbage collection. And we can handle circular data structures but Bornat assumed that data structures are acyclic. The inductive definitions of data structures are much more concise in our approach than those in Bornat's approach.

Advanced type systems have been used to check pointer programs for safety. For instance, Smith, Walker and Morrisett presented alias types [28], the main feature of the new type system is a collection of aliasing constraints. Aliasing constraints describe the shape of the store and every function uses them to specify the store that it expects. If the current store does not conform to the constraints specified, then the type system ensures that the function cannot be called. Alias types very much resemble the fragment of separation logic containing the empty formula, the pointer-to predicate, and separating conjunction. It was also designed for low-level programs. The main difference between the program logics and the type systems is that type systems support better inference while the logics include a wider variety of connectives and more sophisticated constraint systems, and therefore, are much more expressive.

8 Conclusion

In this paper, we present an improved design of the pointer logic, a logic system which can be used to do precise analysis on pointer programs. It can be used to verify whether or not a pointer program satisfies the side conditions of PointerC typing rules, and support the construction of safety proof and other properties proof of pointer programs in PointerC. By now, pointer logic is mainly used to verify programs concerning data structures, in which equality of pointers is well-regulated, such as singly-linked list, circular doubly-linked list and binary tree. Structures like DAG (directed acyclic graph) and other graph, in which equality of pointers is uncertain, is unsupported. But it is our on-going work to add verification support of such structures.

Acknowledgments.

We would like to thank the Professor Shao Zhong for constructive suggestions on a draft of this paper. This research is based on work supported in part by grants from National Natural Science Foundation of China under Grant No.60673126 and No.90718026 and Intel China Research Center. Any opinions, findings and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] G. Necula. Proof-carring code, In *Proc.24th ACM Symposium on Principles of Programming Languages*, pages 106-119, Jan 1997.

[2] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85-97, Jan 1998.

[3] R. Harper and F. Pfenning. Type refinements, *project proposal*, NSF Grant CCR-0204248, Nov 2001.

[4] X. Feng, Z. Shao, A. Vaynberg, S. Xiang and Z. Ni. Modular Verification of Assembly Code with Stack-Based Control Abstractions. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 401-414, June 2006.

[5] X. Y. Feng, Z. Z. Ni, Z. Shao, and Y. Guo. An Open Framework for Foundational Proof-Carrying Code. I*n Proc. 2007 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07), Nice, France*, pages 67-78, January 2007.

[6] H. Xi. Applied type system (extended abstract). In *the post-workshop proceedings of TYPES 2003*, volume 3085 of LNCS, pages 394-408. Springer-Verlag, 2004.

[7] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333-344, June 1998.

[8] Y. Y. Chen, L. Ge, B. J. Hua, Z. P. Li and C. Liu. Design of a Certifying Compiler Supporting Proof of Program Safety. In *Proceedings of 1st IEEE IFIP International Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, pages 117-126, June 2007.

[9] Y. Y. Chen, L. Ge, B. J. Hua, Z. P. Li, C. Liu and Z. F. Wang. A Pointer Logic and Certifying Compiler.*Frontiers of Computer Science in China*.1(3):297-312, July 2007.

[10] Baojian Hua, Yiyun Chen, Zhaopeng Li, Zhifang Wang and Lin Ge. Design and Proof of a Safe Programming Language PointerC. *Chinese Journal of Computers*,31(4):556-564, April 2008

[11] Coq Development Team. The Coq proof assistant reference manual. Coq release v8.1, 2006.

[12] Z. P. Li, Y. Y. Chen, B. J. Hua, and Z. F. Wang. A Revised Pointer Logic for Verification of Pointer Programs (*Extended Version*). Available at: http://ssg.ustcsz.edu.cn/lss/papers/.

[13] B. J. Hua, Y. Y. Chen, L. Ge, and Z. F. Wang. The PointerC programming language specification. (*Technical Report*) Available at: http://ssg.ustcsz.edu.cn/lss/doc/.

[14] Z. P. Li, and Y. Y. Chen. The relations between separation logic and pointer logic. Technique report, http://ssg.ustcsz.edu.cn/lss/doc/, July 2007.

[15] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the* 17^{th} *Annual IEEE Symposium on Logic in Computer Science*, pages 55-74, July 2002.

[16] J. C. Reynolds. An introduction to separation logic. Lecture notes available at http://www.cs.cmu.edu/ jcr/, Spring 2005.

[17] B. J. Hua. Coq implementation of the soundness proof of the pointer logic. Available at http://ssg.ustcsz.edu.cn/lss/software/, June 2007.

[18] Z. P. Li, B. J. Hua. Coq implementation of the core part of the redesigned pointer logic and examples. Available at http://ssg.ustcsz.edu.cn/lss/software/, July 2008

[19] H. B. M. Jonkers. Abstract storage structures. In De Bakker and Van Vliet, editors, Algorithmic languages, pages 321-343, IFIP, North Holland, 1981.

[20] M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Proceedings of PEPM'03*, pages 55-65, ACM Press, 2003.

[21] M. Rometzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of the* 32^{th} *Annual ACM Symposium on Principles of Programming Languages*, pages 296-309, 2005.

[22] C.A.R. Hoare and J. He. A trace model for pointers and objects. In Rachid Guerraoui, editor, *ECCOP'99 - Object- Oriented Programming, 13th European Conference*, pages 1-17, 1999. LNCS. 1628, Springer.

[23] R. Bornat. Proving pointer programs in Hoare logic. In *Proceedings of the* 5^{th} *International Conference on Mathematics of Program Construction*, pages 102-126, July 03-05, 2000.

[24] R.M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23-50, 1972.

[25] J. M. Morris. A general axiom of assignment. Assignment and linked data structure. A proof of the Schorr-Waite algorithm. In *Theoretical Foundations of Programming Methodology(Proceedings of the 1981 Marktoberdorf Summer School), M. Broy and G. Schmidt(eds)*. Reidel(1982)25-51.

[26] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199(1-2), pages 200-227, May 2005.

[27] J. C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Sixth International Conference on Formal Engineering Methods, LNCS3308*, pages 15-29, November 2004.

[28] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proceedings of the 2000 European Symposium on Programming*, pages 366-381, Berlin Mar. 2000.

A dynamic description logic for the three-level RBAC model

Li Ma[1], Shilong Ma[1], Yuefei Sui[2], Cungen Cao[2], and Jianghua Lv[1]

[1] School of Computer Science, Beihang University,
Beijing 100191, China
[2] Key Laboratory of Intelligent Information Processing,
Institute of Computing Technology, Chinese Academy of Sciences,
Beijing 100190, China

Abstract: The Role-based access control (RBAC) model is a promising alternative to traditional discretionary and mandatory access controls. To formalize the RBAC model, the administration of RBAC should be separated from its use for controlling access to data and other resources, where the former contains authorities to administrative actions and the latter, which we call the management of permissions, contains various RBAC components that can be modified by administration actions. Thus, we can view the RBAC in three levels: the authority level, the administration action level and the RBAC component level. To describe the three-level model, a dynamic description logic, called $\mathcal{DDL}_{RBAC}$, is proposed, which is designed for 1) static specifications of administration action authorities, 2) static specifications of administration actions, and 3) static specifications of the regular RBAC components and the dynamic specifications of components changed by actions. To characterize the dynamic feature of RBAC, a new semantic of modality $[a]$ is given, which is different from that of the modal description logic and propositional description logic. A formal description of the whole RBAC is then represented with $\mathcal{DDL}_{RBAC}$.

Keywords: RBAC, role-based administration, modal description logic, action, possible world semantics.

1. Introduction

Role-based access control (RBAC) is a policy-neutral and flexible access control model as a promising alternative to traditional discretionary and mandatory access controls. In RBAC, there are two kinds of management: the management of permissions that controls the user's access to system resources, and the other is the management of RBAC itself. These two kinds of management can be showed in many RBAC models, such as RBAC96 model [1], ARBAC97 model

[2], NIST RBAC model [3], and lately, ANSI standard [4]. RBAC96 discusses the basic components of RBAC, and ARBAC97 gives an administrative model based on RBAC96. NIST RBAC model and ANSI standard further give the precise definition of the basic components and constraints of RBAC, and add in them the administration functions and query functions that should be included in ordinary RBAC systems.

In the management of permissions, various RBAC components, such as user assignment, can be modified to meet the need of changing security policies. And the management is enforced by the configuration of various RBAC components according to the security policies of organizations, so when the security policies changes, the deployment should be changed accordingly. Administration of RBAC itself is very important, and must be carefully controlled to ensure that policy does not drift away from its original objectives[2]. Therefore, both of these two management are critical to the whole RBAC model. We need to unify them into one RBAC model. And also we notice that the permission management and the administration of RBAC itself should be described in different levels, because the former is dynamic and can be changed by administration actions, such as *assign*[2], and the latter is relatively static and unchanged.

However, the existing RBAC models do not completely express this meaning. For example, though ARBAC97 model contains permission management and the administration of RBAC itself, it lacks the representation of the logic relation between the two levels. And NIST RBAC or ANSI RBAC doesn't incorporate the administration of RBAC into its reference RBAC model, which brings about some different understandings to the RBAC model. For example, the notion session is a special concept in RBAC, which can be used to achieving least-privilege. We also noticed that both ordinary users and administrators should create sessions to access resources, which means the session corresponds with actions. Li *et al.* argue that the notion of session should be removed from the Core RBAC model for some management system, such as ESM, has no permissions to be controlled by sessions[5]. While one of the authors of ANSI RBAC model, Ferraiolo, think when a user login a target system, he or she creates a local session within a security context to get assigned roles which is happened in a session of such ESM systems[6]. In fact, both the permission management and the administration of RBAC contain the session notion, while Li's argument is on the permission management level or RBAC component level, and Ferraiolo's explanation is on the administration level, which means the lack of a unified model may bring confusion to some concepts or properties of RBAC.

To get a unified RBAC model and further describe the dynamic properties

and static properties of RBAC, we can view the whole RBAC in three levels of (1) the authority level to describe all of the administration components, such as administration roles, administration permissions, (2) the administration action level to bring the administration and basic RBAC components together, (3) the basic RBAC components level about regular roles and permissions, which is dynamic relative to the first level and can be modified by the second level.

For formalizing RBAC, there are many logical-based approaches, including description logic approaches. However, these description logic approaches give formal descriptions to the aspect only related to the permission management in RBAC [7,8]. We still need to give a formal description of the whole RBAC including the administration of RBAC and the management of permissions.

To describe the three levels of RBAC, we propose a dynamic description logic for RBAC, called $\mathcal{DDL}_{RBAC}$, which consists of three parts 1) the static description for authority specifications of administration actions, 2) the static description for administration actions, and 3) the static description for the RBAC components and the dynamic description for the changing of components by actions. We take action a as a modality $[a]$ and use $[a]\varphi$ to represent the result that a acts on φ , where φ is the statement about RBAC components in $\mathcal{DDL}_{RBAC}$. However, this reading of $[a]$ is different from that of traditional dynamic logics, such as the modal reading of the modal description logic [9,10] or the propositional dynamic logic [11,12], and the programming reading of the quantified dynamic logic [11,12]. We give a detail explanation in section 3.

The paper is organized as follows. The next section gives the three levels of RBAC; the third section defines the logical language, syntax and semantics of $\mathcal{DDL}_{RBAC}$; the fourth section gives a $\mathcal{DDL}_{RBAC}$ model for RBAC and some examples; and the last section concludes the paper.

2. RBAC analysis

In this section, we first introduce a three-level RBAC, and then describe the dynamic properties of RBAC, where we also give an analysis to the actions.

2.1 Three-level RBAC

RBAC is a flexible and policy-neutral access control technology. The policy that is enforced is a consequence of the detailed configuration of various RBAC components. Administration of RBAC is also very important to ensure the security policies not drift away from their original objectives [2]. RBAC96 model illustrates a general family of RBAC models including regular roles and

permissions that regulate access to data and resources, and administrative roles and permissions. This section will first review the model, and then present a three-level RBAC.

Definition 2.1 RBAC96 model [1,2], $\mathcal{M}$, is a 16-tuple

$$(U, R, AR, P, AP, S, UA, AUA, PA, APA, RH, ARH, \\ user, roles, permissions, \Delta'),$$

where

(1) U is a set of users, including regular users and administrative users;

(2) R and AR are disjoint sets of (regular) roles and administrative roles;

(3) P and AP are disjoint sets of (regular) permissions and administrative permissions;

(4) S is a set of sessions;

(5) $UA \subseteq U \times R$, user to role assignment relation (if $(u, r) \in UA$ then u is called a member of r); $AUA \subseteq U \times AR$, user to administrative role assignment relation (if $(u, ar) \in AUA$ then u is called a member of ar);

(6) $PA \subseteq P \times R$, permission to role assignment relation; $APA \subseteq AP \times AR$, permission to administrative role assignment relation;

(7) $RH \subseteq R \times R$, partially ordered role hierarchy; $ARH \subseteq AR \times AR$, partially ordered administrative role hierarchy;

(8) $user, roles$ and $permissions$ are functions such that

(8.1) $user : S \to U$ maps each session s to a single user (which does not change);

(8.2) $roles : S \to 2^{R \cup AR}$ maps each session s to a set $roles(s) \subseteq \{r : \exists r' \geq r((user(s), r') \in UA \cup AUA)\}$ (which can change with time);

(8.3) $permissions : S \to 2^{P \cup AP}$ maps each session s to a set of permissions $\bigcup_{r \in roles(s)} \{p : \exists r'' \leq r((p, r'') \in PA \cup APA)\}$;

(9) Δ' is a collection of constraints stipulating which values of the various components enumerated above are allowed or forbidden.

Remark. To describe the dynamic aspect of RBAC itself, we assume that regular roles and permissions can be modified by administrative actions, while administrative roles and permissions are relatively static and unchanged since we do not consider the administration of administrative roles and permissions in this paper. Thus, we can define a basic RBAC model as a state of regular roles and permissions by excluding administrative roles and permissions.

Definition 2.2 Basic RBAC model, $\mathcal{S}$, is a 11-tuple

$$(U, R, P, S, UA, PA, RH, user, roles, permissions, \Delta').$$

□

The management of RBAC can be viewed as the ability to modify the basic RBAC components, which involves the following administration actions [2]: 1) creation and deletion of roles, 2) creation and deletion of permissions, 3) assignment of permissions to roles and their removal, 4) creation and deletion of users, 5) assignment of users to roles and their removal, 6) definition and maintainence of the role hierarchy, 7) definition and maintainence of constraints. For the simplicity of discussion, we only consider two actions in RBAC: *assign* that grants users to roles and *revoke* that removes users from roles. Obviously, these two actions can only change the components UA.

The ARBAC97 model gives an explicitly definition of the administrative permissions by using *can*-authority relations[2]. For example, the administrative permissions that authorize *assign* and *revoke* are defined in the following authority relations:

• $can\_assign \subseteq AR \times CR \times 2^R$, where CR is a set of prerequisite conditions, consisting of the boolean expressions $\mathbf{t}$ of the following form:

$$\mathbf{t} = \mathbf{r}|\bar{\mathbf{r}}|\mathbf{t}_1 \wedge \mathbf{t}_2|\mathbf{t}_1 \vee \mathbf{t}_2,$$

where $\mathbf{r}$ is a role and $\bar{\mathbf{r}}$ is the negation of $\mathbf{r}$. The meaning of $can\_assign(x, y, X)$ is that a member of the administrative role x (or a member of an administrative role that is senior to x) *can* assign a user whose current membership, or nonmembership, in regular roles satisfies the prerequisite condition y to be a member of regular roles in X;

• $can\_revoke \subseteq AR \times 2^R$. The meaning of $can\_revoke(x, Y)$ is that a member of the administrative role x (or a member of an administrative role that is senior to x) *can* revoke membership of a user from any regular role $y \in Y$.

Hence, we can describe RBAC in three levels: 1) The first level is about administrative users AU, administrative permissions AP, administrative roles AR, administrative role hierarchies ARH, administrative permissions assignment APA, and administrative user assignment AUA. In this level, AP and APA illustrate authorities to administrative roles, and the explicitly definition of AP and APA is the authority relation, such as $can\_assign(x, y, X)$. Thus, this level contains the following elements: AU, AUA, AR, ARH and *can*-authorities. Since we do not consider the administration over administrative roles and permissions, all of the components in this level can be static and unchanged; 2)

the second level is about administrative actions, such as *assign* and *revoke*, which should be compatible with *can_assign* and *can_revoke*, respectively, to show they are authorized; 3) the third level is about various RBAC components related to regular roles and permissions, including U, R, P, UA, PA, RH, S, *users*, *roles*, *permissions*, and constraints. All of these components can be modified by administrative actions.

2.2 A logical analysis to the administrative actions

In ARBAC97 model, the administration actions must be compatible with authority rules. In order to represent this property and the effect of those actions, we give a definition of compatibility and action rules in this section.

To describe the relation between actions, such as *assign* and *revoke*, and authority rules, we use $assignedby(x)$, $assignedto(u)$ and $assigned(r)$ as attributes to describe *assign*, i.e., the member of x executes action *assign*, and role r is assigned to user u, respectively; and use $revokedby(x)$ and $revokedwith(r)$ as attributes to describe *revoke*, i.e., the member of x executes action *revoke*, and user's role r is revoked. Thus, to simplify the description, we use $a(assignedby(x), assignedto(u), assigned(r))$ to represent an assigning action, and $b(revokedby(x), revokedwith(r))$ to represent a revoking action.

Definition 2.3 An action $a(assignedby(x), assignedto(u), assigned(r))$ is compatible with a control $\theta = can\_assign(ar, \varphi, X)$, denoted by $\theta \vdash a$, if

(i) $x \geq ar$; (ii) $u \models \varphi$, and (iii) $r \in X$, where $X \subseteq R$.

Definition 2.4 An action $b(revokedby(x), revoked(r))$ is compatible with a control $\theta = can\_revoke(ar, Y)$, denoted by $\theta \vdash b$, if

(iv) $x \geq ar$; and (v) $r \in Y$, where $Y \subseteq R$.

Definition 2.5 Given a user u, u satisfies t, denoted by $u \models t$, if:

1)$\exists r'(r \leq r' \wedge (u, r') \in UA))$, if $t = r$;

2)$\forall r'(r \leq r' \rightarrow (u, r') \notin UA)$, if $t = \bar{r}$;

3)$u \models t_1 \vee u \models t_2$, if $t = t_1 \vee t_2$;

4)$u \models t_1 \wedge u \models t_2$, if $t = t_1 \wedge t_2$.

The conditions and effects that actions should be satisfied can be described as follows:

(1) Assignment:

$$((x \geq ar) \wedge (u \models t) \wedge (r \in X)) \rightarrow [a(assignedby(x), assignedto(u), assigned(r))]((u, r) \in UA);$$

(2) Revocation:

◦ Weak revocation: for any x,

$$((x \geq ar) \wedge (r \in Y)) \rightarrow [b(revokedby(x), revokedwith(r))]((u,r) \notin UA);$$

◦ Strong revocation: for any x,

$$((x \geq ar) \wedge r \in Y) \rightarrow [b(revokedby(x), revokedwith(r))] \\ ((u,r) \notin UA \wedge \forall r' \geq r((u,r') \notin UA)).$$

Given a partial order $\leq$ on R, we have a structure $\mathcal{R} = (R, \leq)$. Given a user $u \in U$, we have a sub-structure $\mathcal{R}_u = (\mathbf{UA}(u), \leq')$, where

• $\mathbf{UA}(u) = \{r \in R : (u,r) \in UA\} \cup \{\bar{r} : (u,\bar{r}) \in UA\}$; and

• $(\leq)' = \leq\restriction \mathbf{UA}(u)$.($\restriction$ is to represent that $\leq$' is limited in the range of $\mathbf{UA}(u)$).

Let $Th(\mathbf{UA}(u))$ be the default closure of $\mathbf{UA}(u)$, that is, for any $r \in R$, if there is a $r' \in R$ such that $r' \in \mathbf{UA}(u)$, $r \leq r'$, and $\bar{r} \notin \mathbf{UA}(u)$, then $r \in Th(\mathbf{UA}(u))$. Thus, if $r \in \mathbf{UA}(u)$, then u is the explicit member of r; and if $r \in Th(\mathbf{UA}(u))$, then u can be the implicit member of r.

Then we can have action rule as follows:

(1) Given an action $a(assignedby(x), assignedto(u), assigned(r))$ compatible with $can\_assign(ar, t, X)$, i.e., $x \geq ar$, $u \models t$, $r \in X$, we have that

$$[a]\mathbf{UA}(u) = \mathbf{UA}(u) \cup \{r\};$$

(2) Given an action $b(revokedby(x), revokedwith(r))$ that is compatible with $can\_revoke(ar, Y)$, i.e., for any u, $x \geq ar$, $r \in Y$,

a) for weak revocation, we have that

$$[b]\mathbf{UA}(u) = \mathbf{UA}(u) - \{r|(u,r) \in UA\};$$

b) for strong revocation, we have that

$$[b]\mathbf{UA}(u) = \mathbf{UA}(u) - (\{r|(u,r) \in UA\} \cup \{r'|r' \geq r, (u,r') \in UA\}).$$

3. The Dynamic Description Logic $\mathcal{DDL}_{RBAC}$

The dynamic description logic proposed in this paper is based on the basic description logic language $\mathcal{ALC}$ [13,14]. Description logics are originally designed for representing static knowledge. To describe the dynamic properties of RBAC, we can extend the description logic $\mathcal{ALC}$ with dynamic logics to get a dynamic

description logic for RBAC, called $\mathcal{DDL}_{RBAC}$, *inwhich* Action a can be taken as a modality $[a]$.

Administration actions should be compatible with their authorities on the higher level, so here to describe these actions, we can take the conditions that should meet the requirement of authorities as the attributes of actions. Thus, we can introduce action concept and action role. Accordingly, the ordinary concept and role can be called object concept and object roles. In $\mathcal{DDL}_{RBAC}$, the execution of an action can be taken as an individual, we describe the change of object by actions as a modality act on the object statements. So we use a dynamic way to describe actions which is fit for RBAC.

In this section, we will first introduce the syntax and semantics of $\mathcal{DDL}_{RBAC}$, and then point out the difference with other dynamic logics.

3.1. Syntax of $\mathcal{DDL}_{RBAC}$

The logical language L for $\mathcal{DDL}_{RBAC}$ contains the following symbols:

- object (constant) symbols: $c_0, c_1, ...$;
- atomic object concept symbols: $A_0, A_1, ...$;
- object role symbols: $R_0, R_1, ...$;
- object concept constructors: $\neg, \sqcap, \forall$;
- statement constructors: $\sqsubseteq$;
- atomic action symbols: $a_0, a_1, ...$;
- atomic action concept symbols: $B_0, B_1, ...$;
- action role symbols: $S_0, S_1, ...$;
- action constructors: ; , and
- logical connectives: $\neg, \rightarrow$.

Let $\mathcal{A}$ be the set of all the atomic action symbols.

Remark. To simplify the discussion and the logical system, we consider only one action constructor, which is enough for the example we shall give in the following section.

□

Definition 3.1 An object concept C is defined as follows:

$$C = A|\neg C|C_1 \sqcap C_2|\forall R.C|[\alpha]C,$$

where α is an action, defined as follows:

$$\alpha = a|\alpha_1; \alpha_2;$$

Definition 3.2 An action concept D is defined as follows:

$$D = B|\neg D|D_1 \sqcap D_2.$$

Remark. If the interpretation of object concepts is invariant with respect to the possible worlds, then we can define $\forall S.C$ as an action concept, where S is an action role and C is an object concept.

□

Definition 3.3 An object statement φ is defined as follows:

$$\varphi = C(c)|R(c_1, c_2)|C_1 \sqsubseteq C_2|\neg\varphi|\varphi_1 \rightarrow \varphi_2|[\alpha]\varphi_3,$$

where φ_3 is a statement containing no modality.

Definition 3.4 An action statement ψ is defined as follows:

$$\psi = B(a)|S(a, c)|B_1 \sqsubseteq B_2|\neg\psi|\psi_1 \rightarrow \psi_2.$$

Definition 3.5 The knowledge base of $\mathcal{DDL}_{RBAC}$ is $KB = (TBox, ABox, RBox)$, where $ABox_o$ is a set of atomic statements; $TBox_o$ is a set of non-atomic statements containing no modalities; and $RBox_o$ is a set of the statements of form $\varphi \rightarrow [\alpha]\psi$, where φ,ψ contain no modalities;

Remark. If we use $\mathcal{DDL}_{RBAC}$ to describe static knowledge, the $RBox$ is an empty set, then $KB = (TBox, ABox)$.

3.2. Semantics of $\mathcal{DDL}_{RBAC}$

Definition 3.6 A model M for the language of $\mathcal{DDL}_{RBAC}$ is a 5-tuple $(W, \{R_a : a \in \mathcal{A}\}, \Delta, \Sigma, I)$, where

- W is a set of possible worlds;
- for each $a \in \mathcal{A}, R_a \subseteq W \times W$ is the accessibility relation for action a, such that for any $w \in W$, there is at most one $w' \in W$ with $(w, w') \in R_a$;
- Δ is a non-empty universe;
- Σ is a non-empty universe such that $\Sigma \cap \Delta = \emptyset$;
- I is an interpretation such that
  - for each constant symbol $c, I(c) \in \Delta$;
  - for each possible world $w \in W$ and atomic concept symbol $A, I(A, w) \subseteq \Delta$, satisfying that for any $w' \in W$ with $(w, w') \in R_a$ for some $a \in \mathcal{A}, I(A, w) \triangle I(A, w')$ is finite, where $\triangle$ is the symmetric difference, i.e., $X \triangle Y = (X - Y) \cup (Y - X)$, and $I(A, w) - I(A, w') = \{x \in \Delta : x \in I(A, w), x \notin I(A, w')\}$;

– for each possible world $w \in W$ and role symbol $R, I(R,w) \subseteq \Delta^2$, satisfying that for any $w' \in W$ with $(w,w') \in R_a$ for some $a \in \mathcal{A}, I(R,w) \triangle I(R,w')$ is finite, where $I(R,w) - I(R,w') = \{(x,y) \in \Delta^2 : (x,y) \in I(R,w), (x,y) \notin I(R,w')\}$;

Remark. In this possible world semantics, action a may change one state to another state such that the symmetric difference of the two states is finite. Thus the reading of $[a]$ is different from the modal reading and the programming reading. For the detail comparison of the three readings please see subsection **3.3**.

□

– for each atomic action concept $B, I(B) \subseteq \Sigma$;

– for each atomic action role $S, I(S) \subseteq \Sigma \times \Delta$;

Definition 3.7 The interpretation $C^{I,w}$ of an object concept C at possible world w is defined as follows:

$$C^{I,w} = \begin{cases} I(A,w) & \text{if } C = A \\ \Delta - C_1^{I,w} & \text{if } C = \neg C_1 \\ C_1^{I,w} \cap C_2^{I,w} & \text{if } C = C_1 \sqcap C_2 \\ \{x \in \Delta : \mathbf{A}y((x,y) \in R \Rightarrow y \in C_1^{I,w})\} & \text{if } C = \forall R.C_1 \\ \{x \in \Delta : \mathbf{A}w'((w,w') \in R_\alpha \Rightarrow x \in C_2^{I,w'})\} & \text{if } C = [\alpha]C_2, \end{cases}$$

where

$$R_\alpha = \begin{cases} R_a & \text{if } \alpha = a \\ R_{\alpha_1} \circ R_{\alpha_2} & \text{if } \alpha = \alpha_1; \alpha_2, \end{cases}$$

where $R \circ R' = \{(x,y) : \mathbf{E}z((x,z) \in R \& (z,y) \in R')\}$.

Remark. (1) The interpretation $I(c)$ of constant symbol c is independent of possible worlds;

(2) Under the assumption that there is at most one $w' \in W$ with $(w,w') \in R_a$, we use w_a to denote the unique w', and the interpretation $([a]C_2)^{I,w}$ is reduced to C_2^{I,w_a}.

□

Definition 3.8 The interpretation D^I of an action concept D is defined as follows:

$$D^I = \begin{cases} I(B) & \text{if } D = B \\ \Sigma - D_1^I & \text{if } D = \neg D_1 \\ D_1^I \cap D_2^I & \text{if } D = D_1 \sqcap D_2. \end{cases}$$

Definition 3.9 The truth-value $M, w \models \varphi$ of an object statement φ at possible world w is defined as follows:

$$M, w \models \varphi \text{ iff } \begin{cases} I(c) \in C^{I,w} & \text{if } \varphi = C(c) \\ (I(c_1), I(c_2)) \in I(R, w) & \text{if } \varphi = R(c_1, c_2) \\ C_1^{I,w} \subseteq C_2^{I,w} & \text{if } \varphi = C_1 \sqsubseteq C_2 \\ M, w \not\models \varphi_1 & \text{if } \varphi = \neg\varphi_1 \\ M, w \models \varphi_1 \Rightarrow M, w \models \varphi_2 & \text{if } \varphi = \varphi_1 \rightarrow \varphi_2 \\ \mathbf{A}w'((w, w') \in R_\alpha \Rightarrow M, w' \models \varphi_3) & \text{if } \varphi = [\alpha]\varphi_3 \end{cases}$$

Definition 3.10 The truth-value $M \models \psi$ of an action statement ψ is defined as follows:

$$M \models \psi \text{ iff } \begin{cases} I(a) \in I(B) & \text{if } \psi = B(a) \\ (I(a), I(c)) \in I(S) & \text{if } \psi = S(a, c) \\ B_1^I \subseteq B_2^I & \text{if } \psi = B_1 \sqsubseteq B_2 \\ M \not\models \psi_1 & \text{if } \psi = \neg\psi_1 \\ M \models \psi_1 \Rightarrow M \models \psi_2 & \text{if } \psi = \psi_1 \rightarrow \psi_2 \end{cases}$$

Remark. Under the assumption that there is at most one $w' \in W$ with $(w, w') \in R_a$, we use w_a to denote the unique w', and the satisfaction $M, w \models [a]\varphi_3$ is reduced to $M, w_a \models \varphi_3$.

□

We assume that if there is a statement of form $\varphi \rightarrow [\alpha]\psi$ in an *RBox* then there is no other statement of form $\varphi' \rightarrow [\alpha]\psi'$ in *RBox*, where the reading of $\varphi \rightarrow [\alpha]\psi$ is that if φ is true then after executing action α, ψ is true; and α changes the truth-value of no other statements other than φ and ψ. Hence, for any action a and possible worlds $w, w' \in W$, if $(w, w') \in R_a$, then there is a finite set Φ_a^w of atomic statements or the negations of atomic statements such that for any $\varphi \in \Phi_a^w$,

$$w \models \varphi \text{ iff } w' \not\models \varphi;$$

and for any $\varphi \notin \Phi_a^w$,

$$w \models \varphi \text{ iff } w' \models \varphi.$$

Therefore, $\varphi \rightarrow [\alpha]\psi$ in *RBox* can be represented in the following form:

$$\Phi \rightarrow [\alpha]_\varphi \neg\Phi,$$

where φ is the precondition of action α, and for any $w \in W$,

$$\begin{aligned} w \models \Phi \to [\alpha]_\varphi \neg\Phi \quad \text{iff} \quad & w \models \varphi \& \mathbf{A}w'((w,w') \in R_\alpha \Rightarrow \\ & \mathbf{A}\psi \in \Phi(w \models \psi \Leftrightarrow w' \not\models \psi) \\ & \& \mathbf{A}\psi \notin \Phi(w \models \psi \Leftrightarrow w' \models \psi)). \end{aligned}$$

3.3 Comparison

To compare with other dynamic representation, we use $+/-$ to denote the variance/invariance of the interpretation of constant symbols c, concept symbols C, the truth-value of statements φ or assignment of variables x. Then, we have the following table:

	c	C	φ	x
the modal reading	$+/-$	$+$	$+$	$+/-$
the programming reading	$-$	$-$	$+/-$	$+$
the $*$-reading	$-$	$+$	$+$	$\times$

Table 1. Comparison of three readings

where the $*$-reading is the semantics we have given in this paper; $\times$ means that this case is undefined (because there is no variable); and $+/-$ means that the case depends on the semantics we use. For example, in the modal reading of constant $c, I(c,w)$ could be independent of $(-)$ or dependent on $(+)$ the possible world.

Correspondingly, whether there are constraints on two possible worlds one of which is accessible from another via the modalities, is summarized as in the following:

- in the modal reading, there is no constraints on (w,w') if $(w,w') \in R_a$;
- in the programming reading, $\{x : v(x) \neq v'(x)\}$ is finite if v' is accessible from v via some action a;
- in the $*$-reading, for any two possible worlds w, w' with $(w,w') \in R_a$ for some action a, for any atomic concept $C, C^{I,w} \triangle C^{I,w'}$ is finite; and for any role $R, R^{I,w} \triangle R^{I,w'}$ is finite.

4. Representation RBAC in $\mathcal{DDL}_{RBAC}$

This section will give a $\mathcal{DDL}_{RBAC}$ for RBAC, and give some examples from the reference [2].

4.1. $\mathcal{DDL}_{RBAC}$ for RBAC

To describe the three levels of RBAC, we can translate all of the elements of RBAC model, actions and authority into the symbols of $\mathcal{DDL}_{RBAC}$ language. Here we take every authority, such as $can\_assign(x, y, X)$, as an individual such as $\mathbf{can}_{assign}$, and we use atomic role $\mathbf{assignBy}_{can}$ to describe the administrative role x of can_assign, $\mathbf{assignCon}_{can}$ to describe the prerequisite y and $\mathbf{assignWith}_{can}$ to describe the roles in X be assigned by the authority individual. We describe the authority $can\_revoke(x, Y)$ by an individual $\mathbf{can}_{revoke}$, and use atomic roles such as $\mathbf{revokeBy}_{can}$ and $\mathbf{revokeWith}_{can}$ to describe the administrative role x and the roles in Y to be revoked from users, respectively.

Hence we can get a language for the description logic for RBAC consisting of three parts: L_1 for the static authority specification; L_2 for the static description of actions; and L_3 for the static description of RBAC state as well as the dynamic changing, where

- L_1 contains the following symbols:
  - ◇ atomic *can*-authority concept names: $\mathbf{Can}_{assign}, \mathbf{Can}_{revoke}$;
  - ◇ atomic *can*-authority constant names: $\mathbf{can}_{assign_0}, \mathbf{can}_{assign_1}, ...; \mathbf{can}_{revoke_0}, \mathbf{can}_{revoke_1}, ...$;
  - ◇ atomic *can*-authority role names: $\mathbf{assginBy}_{can}$, $\mathbf{assignCon}_{can}$, $\mathbf{assignWith}_{can}$, $\mathbf{revokeBy}_{can}$, $\mathbf{revokeWith}_{can}$;
  - ◇ atomic concept names: **AU**, **AR**, **S**;
  - ◇ atomic role names: **ARH**, **AUA**;
  - ◇ constant names: **au** for each member $au(\in AU)$ of administrative roles; **ar** for each administrative role $ar \in AR$; **s** for each administration session;
  - ◇ prerequisite condition constant name: **t**.
- L_2 contains the following symbols:
  - ◇ atomic action concept names: $\mathbf{Assign}, \mathbf{Revoke}$;

◇ atomic action constant names: $\mathbf{assign}_0, \mathbf{assign}_1, ...; \mathbf{revoke}_0, \mathbf{revoke}_1, ...;$

◇ atomic action role names: **assginBy**, **assignTo**, **assignWith**, **assignCon**; **revokeBy**, **revokeWith**, **revokeFrom**;

and

- L_3 contains the following symbols:

  ◇ atomic concept names: $\mathbf{U}, \mathbf{R}, \mathbf{P}, \mathbf{S}$;

  ◇ constant names: $\mathbf{u}$ for each regular user $u \in U$; $\mathbf{r}$ for each regular role $r \in R$; $\mathbf{p}$ for each regular permission $p \in P$; and $\mathbf{s}$ for each $s \in S$;

  ◇ atomic role names: $\mathbf{UA}, \mathbf{PA}, \mathbf{RH}$, $\mathbf{user}, \mathbf{roles}, \mathbf{permissions}$;

  ◇ a modality $[\mathbf{a}]$ for each action constant name $\mathbf{a}$.

Definition 4.1 A statement φ_1 for L_1 is defined as follows:

$$\begin{aligned}\varphi_1 \quad = \quad & \mathbf{Can}_{assign}(\mathbf{can}_{assign})|\mathbf{Can}_{revoke}(\mathbf{can}_{revoke})|\mathbf{assignBy}_{can}(\mathbf{can}_{assign}, \mathbf{ar}) \\ & |\mathbf{assignCon}_{can}(\mathbf{can}_{assign}, \mathbf{t})|\mathbf{assignWith}_{can}(\mathbf{can}_{assign}, \mathbf{r}) \\ & |\mathbf{revokeBy}_{can}(\mathbf{can}_{revoke}, \mathbf{ar})|\mathbf{revokeWith}_{can}(\mathbf{can}_{revoke}, \mathbf{r}) \\ & |\mathbf{AU}(\mathbf{au})|\mathbf{AR}(\mathbf{ar})|\mathbf{AUA}(\mathbf{au}, \mathbf{ar})|\mathbf{ARH}(\mathbf{ar}, \mathbf{ar}')|\mathbf{S}(\mathbf{s}) \\ & |\neg\varphi_1|\varphi_1^1 \rightarrow \varphi_1^2.\end{aligned}$$

Remark. We use **ARH**(**ar**,**ar**') to denote the partial order **ar**$\leq$**ar**'.

□

Definition 4.2 A statement φ_2 for L_2 is defined as follows:

$$\begin{aligned}\varphi_2 \quad = \quad & \mathbf{Assign}(\mathbf{assign})|\mathbf{Revoke}(\mathbf{revoke}) \\ & |\mathbf{assignBy}(\mathbf{assign}, \mathbf{au})|\mathbf{assignTo}(\mathbf{assign}, \mathbf{u})|\mathbf{assignWith}(\mathbf{assign}, \mathbf{r}) \\ & |\mathbf{assignCon}(\mathbf{u}, \mathbf{r}') \\ & |\mathbf{revokeBy}(\mathbf{revoke}, \mathbf{au})|\mathbf{revokeWith}(\mathbf{revoke}, \mathbf{r})|\mathbf{revokeFrom}(\mathbf{revoke}, \mathbf{u}) \\ & |\neg\varphi_2|\varphi_2^1 \rightarrow \varphi_2^2.\end{aligned}$$

Definition 4.3 A statement φ_3 for L_3 is defined as follows:

$$\begin{aligned}\varphi_3 \quad = \quad & \mathbf{U}(\mathbf{u})|\mathbf{R}(\mathbf{r})|\mathbf{P}(\mathbf{p})|\mathbf{S}(\mathbf{s}) \\ & |\mathbf{UA}(\mathbf{u}, \mathbf{r})|\mathbf{PA}(\mathbf{p}, \mathbf{r})|\mathbf{RH}(\mathbf{r}, \mathbf{r}') \\ & |\mathbf{user}(\mathbf{s}, \mathbf{u})|\mathbf{roles}(\mathbf{s}, \mathbf{r})|\mathbf{permissions}(\mathbf{s}, \mathbf{p}) \\ & |\neg\varphi_3|\varphi_3^1 \rightarrow \varphi_3^2 \\ & |[\mathbf{assign}]\varphi_3|[\mathbf{revoke}]\varphi_3.\end{aligned}$$

Remark. We use **RH**(**r**,**r**') to denote the partial order **r**$\leq$**r**'.

□

To represent conditions (8.2)-(8.3), we need to extend the language to include two role constructors:

$$\sim, \circ,$$

and one role statement constructor:

$$\subseteq,$$

such that if R_1, R_2 are roles then so are $\sim R_1$ and $R_1 \circ R_2$. In such an extended language, (8.2)-(8.3) are represented by

$$\begin{aligned} &\mathbf{roles} \subseteq (\sim (\mathbf{user} \circ \sim \mathbf{UA})) \circ \mathbf{RH}^{-} \\ &\mathbf{permissions} \subseteq \sim (\mathbf{roles} \circ \sim (\mathbf{RH}^{-} \circ \mathbf{PA}^{-})), \end{aligned}$$

respectively.

□

To the whole RBAC, we have the following knowledge bases:

• $KB_1 = (TBox_1, ABox_1)$ is an authority knowledge base, where $TBox_1$ is a set of subsumption axioms; $ABox_1$ is a set of atomic statements about authorities;

• $KB_2 = (TBox_2, ABox_2)$ is an action knowledge base, where $TBox_2$ is a set of axioms about actions, and $ABox_2$ is a set of atomic statements about actions;

• $KB_3 = (TBox_3, ABox_3, RBox_3)$ is a knowledge base for the basic RBAC model $\mathcal{S}$, where $ABox_3$ is a set of atomic statements; $TBox_3$ is a set of non-atomic statements containing no modality; and $RBox_3$ is a set of the statements of form $\varphi \rightarrow [\alpha]\psi$, where φ, ψ contain no modalities.

Remark. In the $TBox$ of a knowledge base, there is a set of the statements specifying the basic properties of the components. For example, in KB_3, there is a set of the statements specifying the disjointness of any two components (e.g., $\mathbf{U} \sqcap \mathbf{R} \equiv \bot$), the domains and ranges of each relations (e.g., $\exists \mathbf{UA}.\mathbf{R} \sqsubseteq \mathbf{U}$ and $\exists \mathbf{UA}^{-}.\mathbf{U} \sqsubseteq \mathbf{R}$) and the functionality of $\mathbf{user}, \mathbf{roles}, \mathbf{permissions}$ (e.g., $(\leq 1\mathbf{user}).\mathbf{U} \sqsubseteq \mathbf{S}$).

□

Definition 4.4 Given knowledge base about actions KB_2, authority knowledge base KB_1, and knowledge base about RBAC model KB_3, we say KB_2 is compatible with KB_1, if

(1) if $\mathbf{Assign}(a) \sqsubseteq KB_2$, and $\mathbf{AU}(\mathbf{au})$, $\mathbf{AR}(\mathbf{ar})$, $\mathbf{AUA}(\mathbf{au}, \mathbf{ar}) \in KB_1$, $\mathbf{U}(\mathbf{u})$, $\mathbf{R}(\mathbf{r}) \in KB_3$, then

$$\mathbf{assignedBy}(a, \mathbf{au}) \in KB_2 \Rightarrow \exists c(\mathbf{Can}_{assign}(c) \in KB_0 \& \mathbf{assignBy}_{can}(c, \mathbf{ar}) \in KB_1);$$
$$\mathbf{assignedWith}(a, \mathbf{r}) \in KB_2 \Rightarrow \exists c(\mathbf{Can}_{assign}(c) \in KB_1 \& \mathbf{assignBy}_{can}(c, \mathbf{ar}) \in KB_1 \& \mathbf{assignWith}_{can}(c, \mathbf{r}) \in KB_1);$$
$$\mathbf{assignedTo}(a, \mathbf{u}) \in KB_2 \Rightarrow \exists c(\mathbf{Can}_{assign}(c) \in KB_1 \& KB_3 \vdash \mathbf{u} \models \mathbf{t}),$$

where c represent an authority individual, and $\mathbf{u} \models \mathbf{t}$ can be defined inductively as follows:

$$\begin{cases} \exists \mathbf{r}'(\mathbf{RH}(\mathbf{r}, \mathbf{r}') \& \mathbf{UA}(\mathbf{u}, \mathbf{r}')) & \text{if } \mathbf{t} = c_{\mathbf{r}} \\ \forall \mathbf{r}'(\mathbf{RH}(\mathbf{r}, \mathbf{r}') \Rightarrow \neg \mathbf{UA}(\mathbf{u}, \mathbf{r}')) & \text{if } \mathbf{t} = \bar{c}_{\mathbf{r}} \\ \mathbf{u} \models \mathbf{t}_1 \text{ or } \mathbf{u} \models \mathbf{t}_2 & \text{if } \mathbf{t} = \mathbf{t}_1 \vee \mathbf{t}_2 \\ \mathbf{u} \models \mathbf{t}_1 \& \mathbf{u} \models \mathbf{t}_2 & \text{if } \mathbf{t} = \mathbf{t}_1 \wedge \mathbf{t}_2 \end{cases}$$

(2) if $\mathbf{Revoke}(a) \in KB_2$, and $\mathbf{AU}(\mathbf{au}), \mathbf{AR}(\mathbf{ar}) \in KB_1$,$\mathbf{U}(\mathbf{u}), \mathbf{R}(\mathbf{r}) \in KB_3$, then

$$\mathbf{revokedBy}(a, \mathbf{au}) \in KB_2 \Rightarrow \exists c(\mathbf{Can}_{revoke}(c) \in KB_1 \& \mathbf{revokeby}_{can}(c, \mathbf{ar}) \in KB_1)$$
$$\mathbf{revokedWith}(a, \mathbf{r}) \in KB_2 \Rightarrow \exists c(\mathbf{Can}_{assign}(c) \in KB_1 \& \mathbf{revokeWith}_{can}(c, \mathbf{r}) \in KB_1).$$

With action $\mathbf{a}$ with statements $\mathbf{assignedBy}(\mathbf{a}, \mathbf{au})$, $\mathbf{assignTo}(\mathbf{a}, \mathbf{u})$ and $\mathbf{assignWith}(\mathbf{a}, \mathbf{r})$, we have the following rules:

$$\neg \mathbf{UA}(\mathbf{u}, \mathbf{r}) \wedge \mathbf{assignedWith}(\mathbf{a}, \mathbf{r}) \rightarrow [\mathbf{a}]_{\varphi} \mathbf{UA}(\mathbf{u}, \mathbf{r}),$$

and with an action $\mathbf{b}$ with statements $\mathbf{revokeBy}(\mathbf{b}, \mathbf{au})$ and $\mathbf{revokeWith}(\mathbf{b}, \mathbf{r})$, we have the following weak revocation rule and strong revocation rule as follows, respectively,

$$\mathbf{UA}(\mathbf{u}, \mathbf{r}) \wedge \mathbf{revokedWith}(\mathbf{b}, \mathbf{r}) \rightarrow [\mathbf{b}] \neg \mathbf{UA}(\mathbf{u}, \mathbf{r}),$$

$$\mathbf{RH}(\mathbf{r}, \mathbf{r}') \wedge \mathbf{UA}(\mathbf{u}, \mathbf{r}') \wedge \mathbf{revokedWith}(\mathbf{b}, \mathbf{r}) \rightarrow [\mathbf{b}] \neg \mathbf{UA}(\mathbf{u}, \mathbf{r}'),$$

where $\mathbf{a}$ and $\mathbf{b}$ are the actions compatible with $can\_assign(ar, \varphi, R)$ and $can\_revoke(ar, R)$, where φ is a prerequisite and R is a role set.

Remark. Here $\mathbf{u}$ and are taken as parameters.

□

At any instant, we have a state description of the whole RBAC of the following form:

$$((KB_1, KB_2), KB_3)$$

where (KB_1, KB_2) does not change, and KB_2 is changing as the state changes.

4.2. Examples

We use examples from [2] (p. 113) to show how the $\mathcal{DDL}_{RBAC}$ can represent the administration of RBAC. The role hierarchies are given by the following diagrams (Fig. 1):

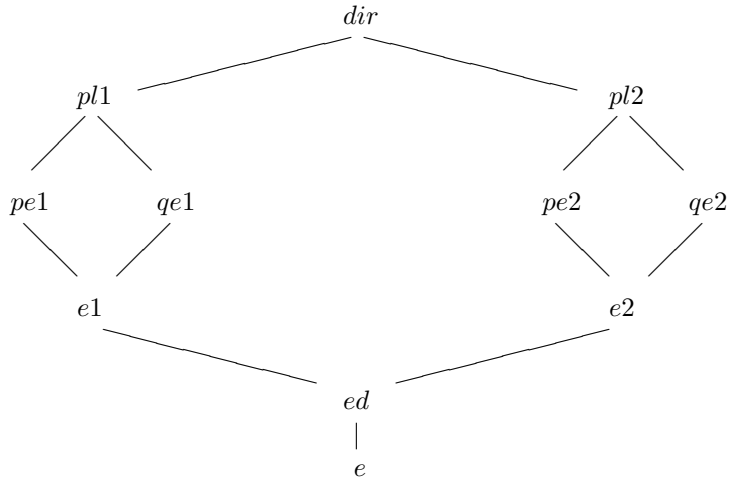


Fig. 1 Role hierarchy example.

The statements for the role hierarchy:

$$\mathbf{RH}(e, ed), \mathbf{RH}(ed, e1), \mathbf{RH}(ed, e2);$$
$$\mathbf{RH}(e1, pe1), \mathbf{RH}(e1, qe1); \mathbf{RH}(e2, pe2), \mathbf{RH}(e2, qe2);$$
$$\mathbf{RH}(pe1, pl1), \mathbf{RH}(qe1, pl1); \mathbf{RH}(pe2, pl2), \mathbf{RH}(qe2, pl2);$$
$$\mathbf{RH}(pl1, dir), \mathbf{RH}(pl2, dir).$$

The administrative hierarchies are given in Fig. 2.

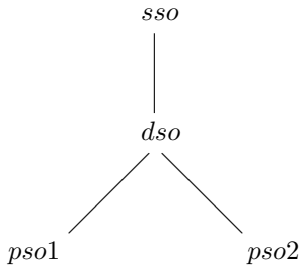


Fig. 2 Administrative hierarchy example.

The statements for the administrative role hierarchy:

$$\mathbf{ARH}(pso1, dso), \mathbf{ARH}(pso2, dso), \mathbf{ARH}(dso, sso).$$

The table for *can_assign* with prerequisite roles

name	*Admin.Role*	*Prereq.Role*	*RoleSet*
c_0	$pso1$	ed	$\{e1, pe1, qe1\}$
c_1	$pso2$	ed	$\{e2, pe2, qe2\}$
c_2	dso	ed	$\{pl1, pl2\}$
c_3	sso	e	$\{ed\}$
c_4	sso	ed	$\{dir\}$

Table 2. *can_assign* with prerequisite roles.

is represented by the following $ABox$:

$$\{\mathbf{Can}_{assign}(c_0), \mathbf{Can}_{assign}(c_1), \mathbf{Can}_{assign}(c_2), \mathbf{Can}_{assign}(c_3), \mathbf{Can}_{assign}(c_4),$$
$$\mathbf{assignBy}_{can}(c_0, pso1), \mathbf{assignCon}_{can}(c_0, ed), \mathbf{assignWith}_{can}(c_0, e1),$$
$$\mathbf{assignWith}_{can}(c_0, pe1), \mathbf{assignWith}_{can}(c_0, qe1);$$
$$\mathbf{assignBy}_{can}(c_1, pso2), \mathbf{assignCon}_{can}(c_1, ed), \mathbf{assignWith}_{can}(c_1, e2),$$
$$\mathbf{assignWith}_{can}(c_1, pe2), \mathbf{assignWith}_{can}(c_1, qe2);$$
$$\mathbf{assignBy}_{can}(c_2, dso), \mathbf{assignCon}_{can}(c_2, ed), \mathbf{assignWith}_{can}(c_2, pl1),$$
$$\mathbf{assignWith}_{can}(c_2, pl2);$$
$$\mathbf{assignBy}_{can}(c_3, sso), \mathbf{assignCon}_{can}(c_3, e), \mathbf{assignWith}_{can}(c_3, ed);$$
$$\mathbf{assignBy}_{can}(c_4, sso), \mathbf{assignCon}_{can}(c_4, ed), \mathbf{assignWith}_{can}(c_4, dir)\}.$$

The table for *can_revoke*

name	*Admin.Role*	*RoleRange*
d_0	$pso1$	$\{e1, pl1\}$
d_1	$pso2$	$\{e2, pl2\}$
d_2	dso	$\{ed, dir\}$
d_3	sso	$\{ed, dir\}$

Table 3. *can_revoke* example.

is represented by the following $ABox$:

$\{\mathbf{Can}_{revoke}(d_0), \mathbf{Can}_{revoke}(d_1), \mathbf{Can}_{revoke}(d_2), \mathbf{Can}_{revoke}(d_3),$
$\mathbf{revokeBy}_{can}(d_0, pso1), \mathbf{revokeWith}_{can}(d_0, e1), \mathbf{revokeWith}_{can}(d_0, pl1);$
$\mathbf{revokeBy}_{can}(d_1, pso2), \mathbf{revokeWith}_{can}(d_1, e2), \mathbf{revokeWith}_{can}(d_1, pl2);$
$\mathbf{revokeBy}_{can}(d_2, dso), \mathbf{revokeWith}_{can}(d_2, ed), \mathbf{revokeWith}_{can}(d_2, dir);$
$\mathbf{revokeBy}_{can}(d_3, sso), \mathbf{revokeWith}_{can}(d_3, ed), \mathbf{revokewith}_{can}(d_3, dir)\}.$

Example 1.(The user-role assignment) Let *Alice* be a member of the $pso1$ role and *Bob* a member of the ed role. *Alice* can assign *Bob* to any of the $e1, pe1$, and $qe1$ roles. Let $\mathbf{can_a}$ be the authority that *Alice* can assign *Bob* to any of the $e1, pe1$ and $qe1$; and let $\mathbf{a}$ be the action that *Alice* assigns *Bob* to $pe1$. The knowledge base is as follows:

$\mathbf{Can}_{assign}(\mathbf{can_a});$
$\mathbf{assignBy}_{can}(\mathbf{can_a}, pso1), \mathbf{assignWith}_{can}(\mathbf{can_a}, e1),$
$\mathbf{assignWith}_{can}(\mathbf{can_a}, pe1), \mathbf{assignWith}_{can}(\mathbf{can_a}, qe1);$
$\mathbf{assignCon}_{can}(\mathbf{can_a}, ed);$
$\mathbf{AU}(Alice), \mathbf{AUA}(Alice, pso1);$
$\mathbf{Assign}(\mathbf{a});$
$\mathbf{assignBy}(\mathbf{a}, Alice), \mathbf{assignWith}(\mathbf{a}, pe1), \mathbf{assignTo}(\mathbf{a}, Bob), \mathbf{assignCon}(\mathbf{a}, ed);$
$\mathbf{U}(Bob), \mathbf{UA}(Bob, ed);$
$\neg\mathbf{UA}(Bob, pe1) \rightarrow [\mathbf{a}]\mathbf{UA}(Bob, pe1).$

Example 2.(The weak user-role revocation). Suppose *Bob* is a member of $pe1$ and $e1$. If *Alice* revokes *Bob*'s membership from $e1$, he continues to be a member of the senior role $pe1$, and therefore can use the permissions of $e1$.

Let $\mathbf{b}$ be the action that *Alice* revokes *Bob*'s membership from $e1$. The knowl-

edge base is as following:

$$\mathbf{Can}_{revoke}(\mathbf{can_b});$$
$$\mathbf{revokeBy}_{can}(\mathbf{can_b}, pso1), \mathbf{revokeWith}_{can}(\mathbf{can_b}, e1);$$
$$\mathbf{AUA}(Alice, pso1);$$
$$\mathbf{Revoke}(\mathbf{b});$$
$$\mathbf{revokeBy}(\mathbf{b}, Alice), \mathbf{revokeWith}(\mathbf{b}, e1), \mathbf{revokeFrom}(\mathbf{b}, Bob);$$
$$\mathbf{UA}(Bob, pe1), \mathbf{UA}(Bob, e1);$$
$$\mathbf{UA}(Bob, e1) \rightarrow [\mathbf{b}]\neg\mathbf{UA}(Bob, e1).$$

However, we have

$$\mathbf{UA}(Bob, pe1) \wedge \mathbf{RH}(e1, pe1) \Rightarrow \mathbf{UA}(Bob, e1).$$

The statement $\mathbf{UA}(Bob, e1)$ means Bob is implied as a member of $e1$, and the revocation is weak.

□

5. Conclusions

We have presented a three-level RBAC model that can logically distinguish the static aspect, such as the components of RBAC administration, from the dynamic aspect, such as the components of permission management in RBAC. We have also proposed a dynamic description logic for RBAC, $\mathcal{DDL}_{RBAC}$, which is designed for 1) static specifications of administration action authorities, 2) static specifications of administration actions, and 3) static specifications of the regular RBAC components and the dynamic specifications of components changed by actions. To characterize administrative actions, a new semantic of modality [a] is given in $\mathcal{DDL}_{RBAC}$, by which action a may change one RBAC state to another state such that the symmetric difference of the two states is finite.

Acknowledgements This work is supported by the Natural Science Foundation (grant nos. 60273019, 60496326, 60573063, and 60573064), and the National 973 Programme (grants no. 2005CB321902).

References:

[1] Sandhu R, Coyne E, Feinstein H, et al. Role-based access control models. IEEE computer, 1996, 29(2): 38-47

[2] Sandhu R, Bhamidipati V, Munawer Q, The ARBAC97 model for role-based administration of roles. ACM Transactions on Information and System Security, 1999, 2(1): 105-135

[3] Ferraiolo D, Sandhu R, Gavrila S, et al. Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security, 2001, 4(3): 224-274

[4] Role-Based Access Control. ANSI INCITS 359-2004, American National Standard for Information Technology, 2004

[5] Li N, Byun J W, Bertino E. A critique of the ANSI standard on role-based access control. IEEE Security and Privacy. 2007, 5(6): 41-49

[6] Ferraiolo D, Kuhn R, Sandhu R. RBAC standard rationale: Comments on "A critique of the ANSI standard on role-based access control", IEEE Security and Privacy. 2007, 5(6): 51-53

[7] Ji G, Tang Y, Jiang Y, et al. A description logic approach to represent and extend RBAC model. In: 1st International Symposium on Pervasive Computing and Applications, 2006, 151-156

[8] Zhao C, Heilili N, Liu S, et al. Representation and reasoning on RBAC: a description logic approach. In: Hung D V, Wirsing M, eds, ICTAC2005, volume 3722 of Lecture Notes of Computer Science. Berlin: Springer, 2005, 381-393

[9] Wolter F, Zakharyaschev M. Dynamic description logics. Advances in Modal Logic, 2000, 2: 449-463

[10] Wolter F, Zakharyaschev M. Satisfiability problem in description logics with modal operators. In: Proceeding of the sixth Conference on Principles of Knowledge Representation and Reasoning, 1998, 512-523

[11] Harel D, Kozen D, Tiuryn J. Dynamic Logic. Foundations of Computing. MIT Press, Cambridge, Massachusetts, 2000

[12] Harel D, Kozen D, Tiuryn J. Dynamic logic. In: Gabbay D, Guenthner F, eds, Handbook of Philosophical Logic (the 2nd edition), 2001, 4: 99-217

[13] Baader F, Calvanese D, McGuinness D L, et al. eds, the Description Logic Handbook, Cambridge University Press, 2002

[14] Baader F, Küsters R, Wolter F. Extensions to description logics, in [13], 226-268

Introduction to GRACE Center

Shinichi Honiden

National Institute of Informatics
Japan

Abstract
GRACE Center is a world-leading software research center in National Institute of Informatics engaged in research, education and practical work in alliances with research organizations in Japan and overseas and as part of industry-academia collaboration. GRACE Center seeks to put in place the foundations of 21st century software, while developing world-class researchers and engineers who will go on to play central roles in the next generation.

Porting GNU Compiler Collection and GNU Binary Utilities for C16X

Le Ton Chanh[1] Le Minh Vu[2] Nguyen Hua Phung[3]

Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology
Ho Chi Minh City, Vietnam

Abstract

When a new machine is released, there must be a new toolchain for it. The toolchain, which includes compiler, assembler, linker, loader, simulator .etc, is used for developing applications and operating systems on the new machine. The task of creating a completely new toolchain is arduous and takes much time. However, with GNU Toolchain, this task is rather simpler. GNU Toolchain provides ways to "port" a new CPU quickly and easily. This paper will explain the process of porting two important factors of GNU Toolchain - GNU Compiler Collection and GNU Binary Utilities, for the C16X Microcontroller.

Keywords: Porting, GCC, Binutils, C16X.

1 Introduction

GNU Toolchain contains several projects: GNU make, GNU Compiler Collection (GCC), GNU Binutils (Binutils), GNU Debugger (GDB), GNU build system (autotools). Among those, GCC, a collection of compilers, and Binutils, a set of tools manipulating machine code such as assembler and linker, are two significant projects. To create a necessary toolchain for a new machine, at least, we need to port GCC and Binutils to the new machine. Documentations [1,2,3,4,5] for porting these projects, however, are rather general and not going into details. In fact, one practical way to port a new machine is to read available ports for other machines and try to apply them to the new port. Some other related works [6,7,8] demonstrate the porting process but concentrating on the task of porting GCC only. The toolchain could not be completed without Binutils, which contains the assembler and the linker.

In this paper, we provide a systematic and understandable guidance for those who want to port GCC and Binutils. We use C16X, a microcontroller introduced

[1] Email: letonchanh@gmail.com
[2] Email: lmvu@cse.hcmut.edu.vn
[3] Email: phung@cse.hcmut.edu.vn

by Infineon, to demonstrate the process of creating a new compiler, assembler and linker for a new machine. Users who want to port a similar architecture to C16X could use this paper as a reference for their work.

The paper is organized as follows. In the next section, we describe briefly the roles of GCC and Binutils. Section III presents the process of porting Binutils while Section IV introduces how to port GCC. We summarize our work in Section V.

2 Overview

Figure 1 gives an overview about the roles of GCC and Binutils. GCC includes a preprocessor that provides a set of services such as file inclusion, macro substitution and conditional compilation. An important component of GCC is a compiler, named gcc, that translates a source code into assembly code. Binutils contains an assembler, named as, that translates assembly code into object code, and a linker, named ld, that allows making an executable file from many object code files. Given an appropriate description of a new machine, these tools can generate and manipulate code of that machine. The next section explains how to give such a description to Binutils.

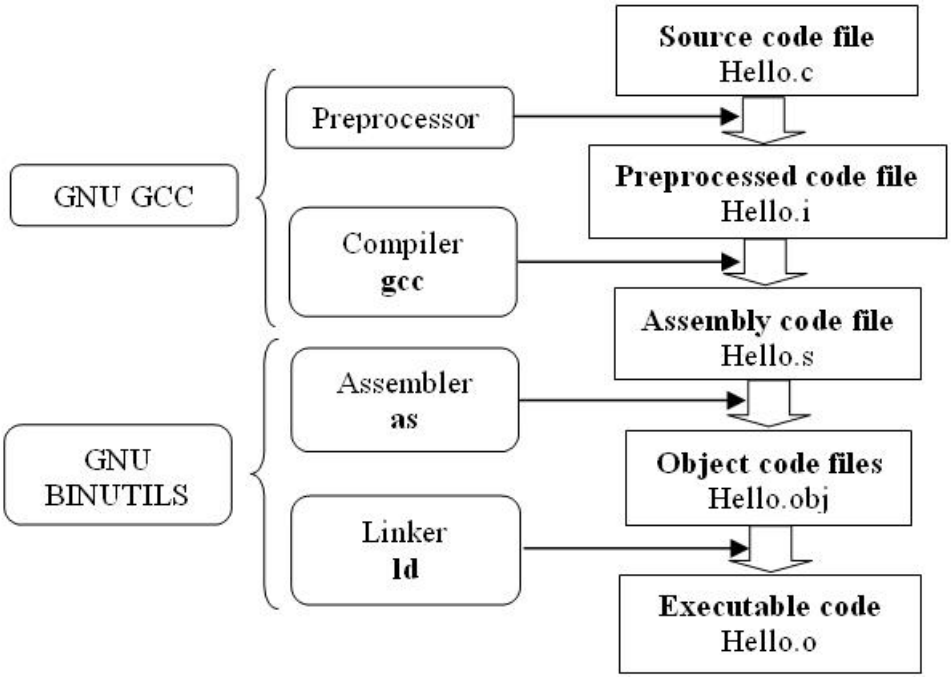


Fig. 1. **The roles of GCC and Binutils**

3 Binutils Porting

Porting GNU Binutils basically consists of porting the followings components: BFD, CGEN, gas and ld.

3.1 BFD Porting

The first component which should be ported is BFD (Binary File Descriptor). This component is a library which provides features to read and write object files, executables, archive files, and core files in any format [1]. These features are used by several other tools such as debugger (gdb), object dumper (objdump), object copier (objcopy), assembler (gas), or linker (ld). Some following new files must be created in directory bfd for porting BFD.

3.1.1 New files for porting BFD

- A new file cpu-CPU.c (e.g., cpu-c16x.c), which defines the information for each machine that this architecture supports. This infomation contains the bit per word, bit per address, bit per byte, architecture, machine...). This file must include three ".h" files: sysdep.h, bfd.h, and libbfd.h.
- A new file elf32-CPU.c (e.g., elf32-c16x.c), which defines the way to map relocation types defined in BFD into those of the ported architecture (i.e., C16X).
- A new header file in the 'include/elf' directory called 'cpu.h' (e.g., c16x.h). This file should define any target specific information which may be needed outside of the BFD code. In particular it should use the START_RELOC_NUMBERS, RELOC_NUMBER, FAKE_RELOC, EMPTY_RELOC and END_RELOC_NUMBERS macros to create a table mapping the number used to identify a relocation to a name describing that relocation.

3.1.2 Adding porting information into existing BFD files

- config.bfd: This file converts a canonical host type into a BFD host type.
- targets.c: This file defines generic target-file-type support for the BFD library.
- archures.c: This file defines the BFD library support routines for architectures.

3.2 CGEN Porting

One goal of CGEN is to describe the CPU in an application independent manner so that program generators can do all the repetitive work of generating code and tables for each CPU that is ported [2]. A CPU description file, that provides some specific types of CPU properties, is required to port CGEN.

3.3 GAS Porting

GAS is an assembler that translates assembly code into object code. Figure 2 describes the working process of GAS. It firstly initializes itself by calling init routines. After that, it repeatedly reads one by one line in assembly code. For each line, GAS performs appropriate actions depending on that the first word is pseudo-op or the instruction. When it finishes reading file, GAS will write the object code into an object file using BFD.

To port GAS, we need to provide the information of the target CPU in some CPU files, tc-CPU.h and tc-CPU.c, located in directory config.

The following macros must be defined in tc-CPU.h file:

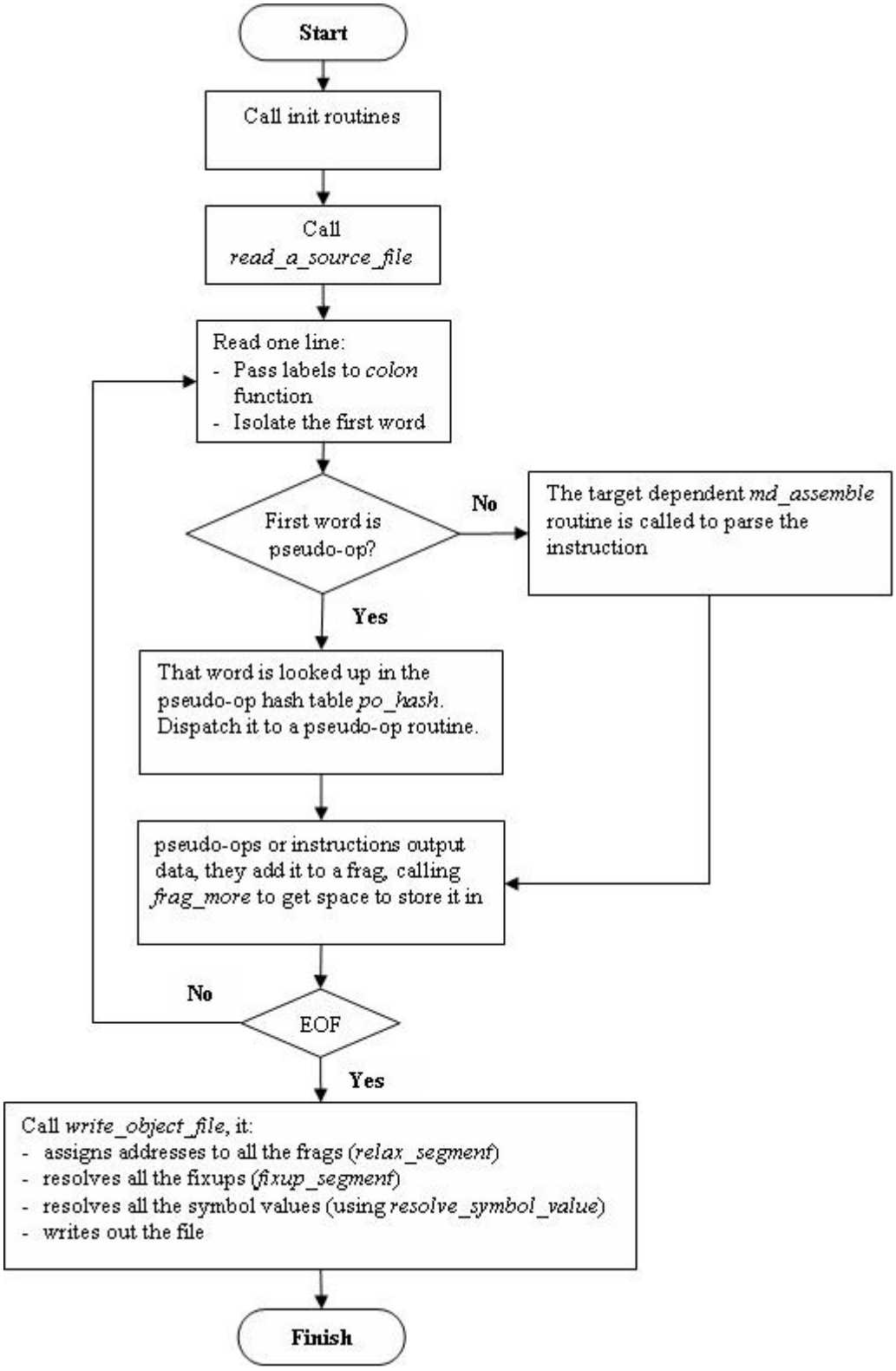


Fig. 2. **The working process of GAS**

- TC_CPU: By convention, this macro should be defined in tc-CPU.h.
- TARGET_FORMAT: This macro is the BFD target name to use when creating the output file. This will normally depend upon the OBJ_FMT macro.
- TARGET_ARCH: This macro is the BFD architecture to pass to bfd_set_arch_mach.
- TARGET_BYTE_BIG_ENDIAN: It should be non-zero if the target is big endian, and zero if the target is little endian.

The following functions must be defined in tc-CPU.c file:

- md_begin: GAS will call this function at the beginning of the assembly, after the command line arguments have been parsed and all the machine independent initializations have been completed.
- md_assemble: This function will be called for each input line which does not contain a pseudo-op. The argument is a null terminated string. The function should assemble the string as an instruction with operands.
- md_section_align: GAS will call this function for each section at the end of the assembly, to permit the CPU backend to adjust the alignment of a section.
- tc_gen_reloc: This function will be called by GAS to generate a reloc. GAS will pass the resulting reloc to bfd_install_relocation.

  Other macros and functions could be found in [3].

3.4 Ld Porting

LD is a linker that takes one or more object files and combines them into a single executable program. For each target, the linker has its specific emulation that replaces the linker default values with the other aspects of the target system. A new emulation, therefore, needs to be created to port LD. The process of creating a new emulation is perfomed by running the script 'genscripts.sh'. More details are given in [4].

4 GCC Porting

4.1 Introduction

4.1.1 The structure of GCC

GCC is the GNU compiler collection that allows to compile many programming languages such as C, C++, JAVA and be able to produce assembly code for dozens of systems. GCC achieves its flexibility with a modular design which is described in Figure 3. It is broken into three parts: the front-end scans and parses the input code of a program, and converts it to GENERIC, a language-independent intermediate representation. The intermediate program then is passed to the language- and system-independent middle-end, which does the majority of the optimizations with the SSA-based GIMPLE language. Finally, the back-end transforms the program into RTL representation, does the final optimizations and generates assembly code.

In the back-end, GCC relies on the RTL-form machine description, which depends on respective target machine. The RTL can be thought of as an assembly

language for an abstract machine that has unlimited number of registers. Be a low-level representation, RTL works well for machine-specific optimizations (for example, register allocation, delay slot optimizations, peepholes, etc) and assembly code generation. The back-end of GCC has some tools, which are tasked with mining the machine description. They extract machine-dependent information and use this information to generate components, which are linked with the components of front-end and middle-end to give the final compiler executable.

4.1.2 Porting GCC process

The porting GCC process is the declaration of the target machine's information for GCC. GCC offers a fairly flexible way for developers to describe their target machine when porting GCC to a new architecture through the pre-defined macros and the instruction patterns. The machine description contains in the following files that belong to the folder <gcc_root>/gcc/config/<machine> (<machine> is the name of machine to which GCC is ported):

- <machine>.md: contains instruction patterns for generating a list of RTL instructions (insn, for short) from parse tree and generating assembly code. It also contains the description of machine-specific optimizations.
- <machine>.h: contains macro definitions of target machine's Application Binary Interface (ABI).
- <machine>-protos.h and <machine>.c: contains the declaration and implementation of the user-defined functions used in
- <machine>.md and <machine>.h.

Some additional files used in linker phase of compilation such as the initial code 'crt0.s' or the libraries in assembly code 'libgcc.s' must be declared in t-<machine> file. The <machine>.md and <machine>.h are the most important files in a machine description and they must be implemented. The following sections will introduce the way to write the <machine.h> and <machine.md> for a machine description.

4.2 Write the machine description (<machine>.md) file

The substance of <machine>.md file is the instruction set of abstract machine written by RTL. GCC provides an available set of standard pattern names. The developers must write a pattern for each instruction that their target machine supports. The declarations for machine-specific optimizations are also written in this file.

Firstly, the presentations of the instructions' operands must be specified. They are used when a single instruction has multiple alternative sets of possible operands. The constraints are representative of those operands in an instruction pattern. They are defined based on the target machine's specification and instruction set. There are many pre-defined standard pattern names in GCC. Only the instruction names that are meaningful for the target machine need writing in <machine>.md file. The implementation of name 'movm' is mandatory (m stands for a two-letter machine mode name, in lowercase). In GCC-C16x, the name 'moveqi' and 'movehi' are im-

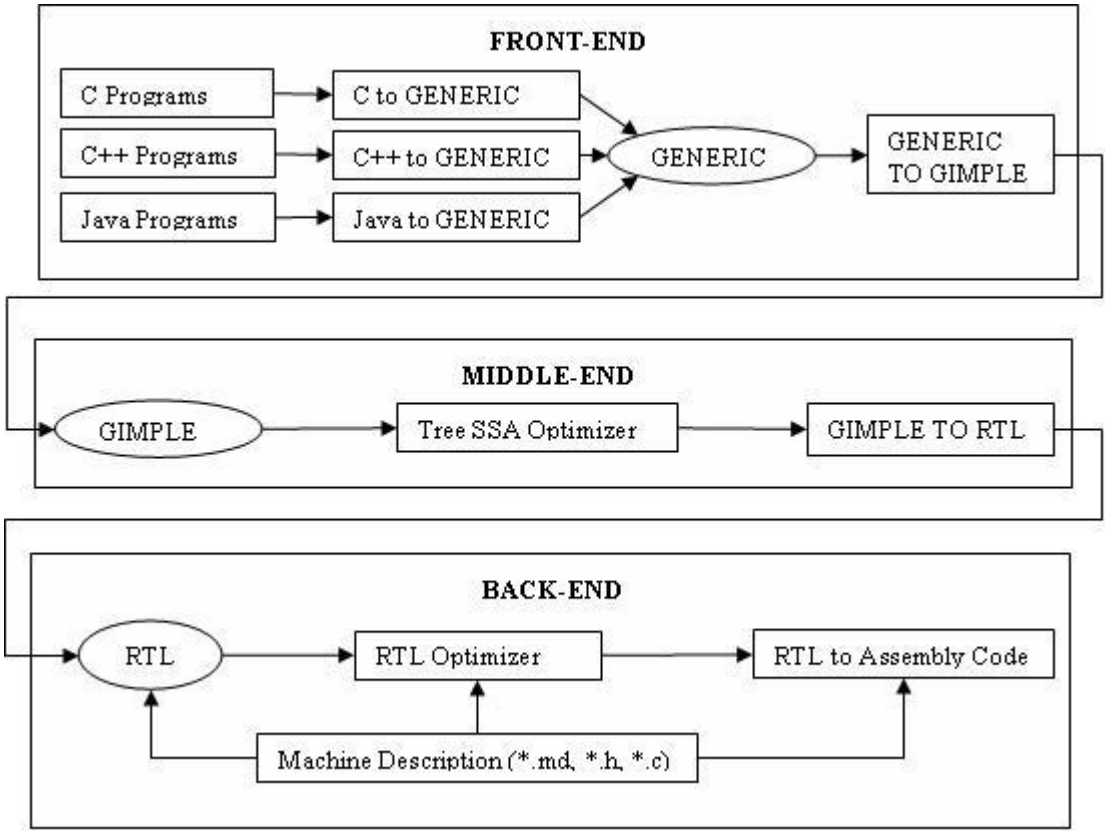


Fig. 3. **Modular design of GCC**

plemented for moving the 8-bit and 16-bit data, respectively. The RTL expression 'define_insn' is used to specify the instruction pattern. It has four or five operands. In which, the RTL template operand is used to define which insns match the particular pattern and the output template operand is used for the generation of assembly code. The 'define_expand' expression generates a sequence of insn expressions for a correlative complex task, which cannot describe by only one 'define_insn' expression.

For optimization purpose, the 'define_split' expressions specify how to split a pattern into many multiple insns for scheduling by 'delay slots'. The 'define_peephole' and 'define_peephole2' expressions contain definitions of machine-specific peephole optimizations.

Most modern processors have a pipeline mechanism and GCC supports the developers to specify this feature by expressions:

```
(define_automaton automata-names)
(define_cpu_unit unit-names [automaton-name])
(define_insn_reservation insn-name default_latency condition regexp)
```

Based on those expressions, the finite state automaton based pipeline hazard recognizer is generated automatically and the instruction scheduler inside GCC uses this automaton for scheduling.

4.3 Specify the ABI of the target machine in the header file (<machine>.h)

The Application Binary Interface (ABI) describes the low-level interface between an application and its libraries, or between components of an application. Normally, the ABI of a target machine includes information about registers, stack layout, calling conventions, passing arguments, etc. GCC has about 500 macros, but the developers do not need to define all of them because the default values of many macros are suitable for various target machines. In reality, only about 150 macros are implemented in the GCC-C16x.

To be able to use the needed macros, the target '<machine>.c' file must declare the global targetm variable, which contains pointers to functions and data relating to the target machine as follows:

```
#include "target.h"
#include "target-def.h"
struct gcc_target targetm = TARGET_INITIALIZER;
```

Besides some simple macros that describe storage layout, data type layout and register usage, some other groups of macros need to be redefined in the GCC-C16x compiler as follows.

4.3.1 Calling conventions macros

In GCC-C16x, jump instructions are used in calling functions. Hence, the return address of functions will be saved on the user stack instead of system stack. This solution helps increase the number of calling recursive functions. In this case, it only depends on the limitation of memory.

The GCC-C16x compiler implements two passing function arguments mechanisms: passing on the stack and passing in the registers. Each of them has its own advantages. Passing on the stack should be used for the functions that have many local variables and have many operations on those variables in their body, so the registers should be reserved for storing these variables. In the contrast case, which the function is only the intermediary for passing arguments between other functions, the speed-up of program will increase if its arguments pass via registers.

For implementing the calling conventions, the developers must define two macros TARGET_ASM_FUNCTION_PROLOGUE, TARGET_ASM_FUNCTION_EPILOGUE with the function prologue and the function epilogue for callee, respectively in C source file '<machine>.c'.

```
#undef TARGET_ASM_FUNCTION_PROLOGUE
#define TARGET_ASM_FUNCTION_PROLOGUE c16x_asm_function_prologue
#undef TARGET_ASM_FUNCTION_EPILOGUE
#define TARGET_ASM_FUNCTION_EPILOGUE c16x_asm_function_epilogue
```

Besides two above files, the c16x_asm_expand_call function must also be implemented. This function is responsible for saving the return address and emits jump instruction to callee's address. It is used by define_insn expression with "*call" name in <machine>.md.

The passing arguments on the stack is the default mechanism in the GCC-C16x compiler. To instruct GCC to pass arguments on stack, the macro ACCUMU-

LATE_OUTGOING_ARGS must be defined as follow:

```
#define ACCUMULATE_OUTGOING_ARGS 1
```

This definition requires GCC to compute the maximum amount of stack space required for outgoing arguments and place it into the variable current_function_outgoing_args_size. The function c16x_asm_function_prologue uses this variable for setting up the space of argument block on the activation record of each function.

For passing arguments in registers, user must insert compiling directive '-mregsparm' in the command line to enable this function. If this mode is enabled, GCC-C16x reserves five registers from R5 to R9 for passing arguments of all functions. If the number of function's arguments is more than five, the remaining arguments will be passed via stack.

The GCC-C16x also has a mixed mode for passing arguments on the stack and in the registers in a same program concurrently. If a function is assigned the attribute 'get_parameters_from_reg' by programmer, it knows that it must receive arguments from prescribed registers. When calling function that has this attribute, the caller will put the parameters in the registers. These things can be achieved by the declaration of the following macros:

```
#define NREGS_FOR_REG_PARM 5
#define FIRST_PARM_REG 5
#define FUNCTION_ARG(CUM,MODE,TYPE,NAME) func_arg(CUM,MODE,TYPE,NAME)
```

The function func_arg is implemented in <machine>.c. The function controls whether an argument is passed in a register, and which register, based on CUM (the summary of all the previous arguments), MODE (the machine mode of the argument) and TYPE (the data type of the argument). If this function returns NULL_RTX, the current argument must be passed on the stack.

The macro INIT_CUMULATIVE_ARGS (CUM, FNTYPE, LIBNAME, FNDECL, INDIRECT) is a C statement for initializing the variable cum for the state at the beginning of the argument list. Based on the current compiling mode or attribute of FNDECL, the compiler initials the correlative value of cum: 0 if passing arguments in registers or (NREGS_FOR_REG_PARM * UNITS_PER_WORD) if passing arguments on the stack.

The macro FUNCTION_ARG_ADVANCE increases the value of cum to advance to the next argument in the argument list.

4.3.2 Macros for defining constraints used in the <machine>.md file

The machine-dependent operand constraint letters used in <machine>.md are defined by the macros CONST_OK_FOR_LETTER¶, for specifying particular ranges of integer values, and REG_CLASS_FROM_LETTER(C), for specifying the register classes.

For two constraints of memory reference A and B, the macro EXTRA_CONSTRAINT(X, C) will be defined with the function c16x_extra_constraint in the <machine>.c file. The function checks whether the operand with constraints A and B matches the form [Rw] and [Rw + #data16], respectively.

4.3.3 Macros for assembly instruction output

Many macros must be defined for output assembly code accurately. The most important macros are PRINT_OPERAND and PRINT_OPERAND_ADDRESS. The macro PRINT_OPERAND prints out operands of instruction in right assembly code form based on constraints operands. Similarly PRINT_OPERAND, the macro PRINT_OPERAND_ADDRESS prints out operands with memory reference constraints.

4.3.4 Macros for adjusting the instruction scheduler

Besides writing RTL expressions for describing pipeline mechanism of machine, many macros should be defined in <machine¡>.h to adjust the instruction scheduler. If the developer decide to use an instruction scheduler in his compiler, the definition of the macro TARGET_SCHED_USE_DFA_PIPELINE_INTERFACE is mandatory.

5 Conclusion

We provided an overview of porting process Binutils and GCC. With those tools, it is easy to make a linker, an assembler and a compiler for a new machine. In fact, porting GCC for a microcontroller like C16X requires around 3500 lines of codes while Binutils port contains more than 4000. We described the details of the components in Binutils: BFD, CGEN, GAS and LD. These components help to make a linker and an assembler through some description files. We also gave the details of porting GCC that help to create a compiler for a new machine.

References

[1] Taylor, I. L., "BFD Internals".

[2] Ellison, B., "CGEN", 2003.

[3] "Assembler Internals", GAS Document.

[4] Bothner, P., and Chamberlain, S., and Taylor, I. L., "A guide to the internals of the GNU linker".

[5] Stallman, R. M., "Using and Porting the GNU Compiler Collection", Free Software Foundation, 2001.

[6] Gunnarsson, H., and Lundqvist, T., "Porting the GNU C Compiler to the Thor Microprocessor", 1995.

[7] Nillsson, H. P., "Porting GCC for Dunces", 2000.

[8] Parthey, J., "Porting the GCC-Backend to a VLIW-Architecture", 2004.

The Third Homomorphism Theorem on Trees
Upward & Downward Leads to Divide-and-Conquer

Akimasa Morihata[†1] Kiminori Matsuzaki[†2] Zhenjiang Hu[‡3]
Masato Takeichi[†4]

†*University of Tokyo, Tokyo, Japan*

‡*National Institute of Informatics, Tokyo, Japan*

Abstract

Parallel programs on lists have been intensively studied. It is well known that divide-and-conquer parallel programs are efficient in the sense that they show good scalability with respect to the number of processors, and associativity provides a good characterization for them. In particular, *the third homomorphism theorem* is a theorem that is not only useful for systematic development of parallel programs on lists also useful for automatic parallelization. The theorem states that if two sequential programs iterate the same list leftward and rightward, respectively, and compute the same value, then there exists a divide-and-conquer parallel program that computes the same value as the sequential programs.
In contrast to rich studies on lists, few studies have been done for characterizing and developing of parallel programs on trees. Naive divide-and-conquer programs, which divide a tree at its root and compute independent subtrees in parallel, show poor scalability with respect to the number of processors when the input tree is ill-balanced, because a bit larger subtree will form a bottleneck.
In this presentation, we develop a method for systematically constructing scalable divide-and-conquer parallel programs on trees. We focus on paths instead of trees so as to utilize rich results on lists; then, we demonstrate that associativity provides good characterization for scalable divide-and-conquer parallel programs on trees. Moreover, we generalize the third homomorphism theorem from lists to trees, in which two sequential programs lead to a scalable divide-and-conquer parallel program. Our results, being generalizations of known results for lists, are generic in the sense that they work well for a large class of trees called polynomial data structures.
This result was published in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Savannah, Georgia, USA, January 21-23, 2009.*

Keywords: Divide-and-conquer algorithms, Parallel programming, Polynomial data structures

[1] Email: morihata@ipl.t.u-tokyo.ac.jp
[2] Email: kmatsu@ipl.t.u-tokyo.ac.jp
[3] Email: hu@nii.ac.jp
[4] Email: takeichi@mist.i.u-tokyo.ac.jp

Algebra of Programming using Dependent Types

Shin-Cheng Mu * Hsiang-Shang Ko † Patrik Jansson ‡

Program derivation is the technique of successively applying formal rules to a specification until one obtains a program. The program thus obtained is correct by construction. In relational program derivation [2], a relational specification is stepwise refined to a functional program by an *algebra of programs*. Meanwhile, type theorists take a complementary approach to program correctness. Modern programming languages deploy advanced type systems that are able to express various correctness properties. This work aims to show, in the dependently typed language Agda [6], how program derivation can be encoded in a type and its proof term. A program and its derivation can thus be written in the same language, and the correctness is guaranteed by the type checker.

As a teaser, Fig. 1 shows a derivation of a sorting algorithm in progress. The type of *sort-der* is a proposition that there exists a function f that, after being lifted to a relation by *fun*, is contained in *ordered*? $\circ$ *permute*, a relation mapping a list to one of its ordered permutations. To prove an existential proposition, we provide a pair of a witness and a proof that the witness satisfies the proposition. The witness is left out as an underline (_), while the proof proceeds by stepwisely refining (through relational inclusion $\sqsupseteq$) the specification. The first step exploits monotonicity of $\circ$ and that *permute* can be expressed as a fold. The second step makes use of relational fold fusion, but the fusion conditions are not given yet. The shaded areas denote *interaction points* — fragments of (proof) code to be completed. The programmer can query Agda for the expected type and the context of the shaded expression. When the proof is completed, an algorithm *isort* is obtained by extracting the witness of the proposition. It is an executable program that is backed by the type system to meet the specification. All is done by exploiting existing the type system and interactive environment of Agda.

Our work aims to be a co-operation between the *squiggolists* and dependently typed programmers that may benefit both sides:

- this is a case study of using the Curry-Howard isomorphism which the squiggolists may appreciate: specifications are expressed in their types, whose proofs (derivations) are checked by the type system. Being able to express derivation *within* the same programming language encourages its use and serves as documentation.

*Institute of Information Science, Academia Sinica, Taiwan

†Department of Computer Science and Information Engineering, National Taiwan University, Taiwan

‡Department of Computer Science and Engineering Chalmers University of Technology & University of Gothenburg, Sweden

```
sort-der  :  ∃ (λf → ordered? ∘ permute ⊒ fun f)
sort-der  =  (_, (  ordered? ∘ permute
                 ⊒⟨  ∘-mono-r permute-is-fold  ⟩
                    ordered? ∘ foldR combine nil
                 ⊒⟨  foldR-fusion-⊒ ordered? { }0 { }1  ⟩
                    { }2 ))

isort  :  List Val → List Val
isort  =  proj₁ sort-der
```

Figure 1: A derivation of insertion sort in progress.

- We modelled a wide range of concepts that often occur in relational program derivation, including relational folds [1], relational division, converse-of-a-function [3]. We have presented several non-trivial derivations, including an optimisation problem, and a relational derivation of quicksort.
- In dependently typed programming it is vital to ensure that a program terminates. To deal with unfolds and hylomorphisms, we allow the programmer to model an unfold as the relational converse of a fold, but demand a proof of *accessibility* [5] before it is refined to a functional unfold. The connection between accessibility and *inductivity* [2] is explained.

The library we have developed, nicknamed AoPA (Algebra of Programming in Agda), is available online [4].

References

[1] R. C. Backhouse et al. Relational catamorphisms. In *IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier, 1991.

[2] R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[3] S.-C. Mu and R. S. Bird. Theory and applications of inverting functions as folds. *Science of Computer Programming*, 51:87–116, 2003.

[4] S.-C. Mu, H.-S. Ko, and P. Jansson. AoPA: Algebra of programming in Agda. `http://www.iis.sinica.edu.tw/~scm/2008/aopa/`, 2008.

[5] B. Nordström. Terminating general recursion. *BIT Numerical Mathematics*, 28(3):605–619, September 1988.

[6] U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers Univ. of Tech., 2007.