Any-time collaborative programming environment and supporting techniques

Fan, Hongfei

2013

Fan, H. (2013). Any-time collaborative programming environment and supporting techniques. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/54902>

<https://doi.org/10.32657/10356/54902>

**ANY-TIME COLLABORATIVE
PROGRAMMING ENVIRONMENT AND
SUPPORTING TECHNIQUES**

FAN HONGFEI

SCHOOL OF COMPUTER ENGINEERING

2013

**ANY-TIME COLLABORATIVE
PROGRAMMING ENVIRONMENT AND
SUPPORTING TECHNIQUES**

FAN HONGFEI

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2013

Acknowledgments

First of all, I would like to express my sincerest gratitude to my supervisor, Professor Sun Chengzheng. Six years ago, he offered me the opportunity to pursue a PhD degree at Nanyang Technological University. For the past five years, he has made every effort to provide me with invaluable guidance and support which have covered every aspect of the research work. I learnt a lot from him, not only specialized skills and knowledge, but also general principles and methods in conducting research work. I believe that my experiences accumulated within these years will benefit me for the rest of my life.

I am deeply indebted to my beloved girlfriend, dearest Liu Jingxian, for her tremendous support, care, understanding, encouragement and love, as well as all the happiness and sweetness she has given to me, every day and night, throughout the past five years. It is Jingxian who gives me the power and keeps me going all the time. When I experienced disappointment and unhappiness, she carried me through the difficult times. I will never forget how we went through days and nights over these years.

I owe a great deal to my parents, Ms. Wang Xia and Mr. Fan Weizhong, who have given me great support and encouragement for everything along the way. I wish to thank my brother, Mr. Fan Chen, who offers valuable advice to me from time to time and encourages me a lot. I extend my gratitude to my grandparents, my uncle and aunt, my family, and Jingxian's parents and family, for their emotional support and care all the time.

I appreciate the support from all members of the research group. Dr. Agustina has provided numerous invaluable suggestions and detailed feedback on this thesis. Dr. Mao Yuqing has shared his experiences in many issues, and contributed valuable ideas on my

research. Dr. Shen Haifeng has provided constructive feedback and suggestions on my work. Miss Xu Yi gives me encouragement and support all the time. I also extend my thanks to all other group members, for their support and interests in my work.

I wish to thank all my friends at NTU, past and present, including Mr. Hao Jianan, Mr. Jin Jiangming, Mr. Li Yusen, Mr. Tang Shanjiang, Mr. Wen Hongkai, Miss Xu Xiang, Mr. Zha Wei, Mr. Zhang Yang, Mr. Zhong Jianlong, and many others, for their companionship and support, as well as the happiness they brought to my life in Singapore.

I extend my gratitude to my old friends, Mr. Ou Jiawei, Ms. Tian Ying, Mr. Wang Fan, Ms. Wen Chanjuan, Mr. Xu Boying, Mr. Zhang Ke, and many others, for their long-term care, support and encouragement. Especially, I have learnt a lot and drawn much inspiration from numerous discussions with Mr. Xu Boying on my research work.

I appreciate the support from the School of Computer Engineering and the Parallel and Distributed Computing Centre at NTU. The laboratory is equipped with good facilities and wonderful environments. In particular, I sincerely thank Ms. Ng-Goh Siew Lai and other technicians for their great support in many aspects.

Furthermore, I wish to extend my gratitude to Dr. Gao Zhen, Mr. He Zongjian, Prof. Jia Jinyuan, Prof. Liu Qin, Dr. Qin Wenzhong, Prof. Zhang Chenxi, Ms. Zhang Di, Mr. Zhu Hongming, and many others, from the School of Software Engineering at Tongji University, for their long-term care, support and encouragement.

In fact, too many people have supported me along the way, and given me the possibility to have the achievement today. I am not able to list all their names, and words cannot truly express my gratitude to them. I will keep all appreciations in my heart forever.

Table of Contents

Acknowledgments	i
Table of Contents.....	iii
List of Figures.....	vii
List of Tables	ix
Abbreviations and Acronyms	x
Abstract.....	xii
Chapter 1. Introduction.....	1
1.1 Collaborative Software Development	1
1.1.1 Collaboration in Software Development	1
1.1.2 Collaborative Activities in Software Development Phases	2
1.2 The Need for Collaborative Programming	5
1.3 Non-Real-Time Collaborative Programming	8
1.4 Real-Time Collaborative Programming	13
1.5 Characteristics, Techniques, and Dynamic Needs for Non-Real-Time and Real-Time Collaborative Programming	20
1.5.1 Characteristics of Non-Real-Time and Real-Time Collaborative Programming	20
1.5.2 Complementary Techniques for Supporting Non-Real-Time and Real-Time Collaborative Programming	22
1.5.3 Dynamic Collaboration Needs at Different Stages of the Programming Process	25
1.6 Any-Time Collaborative Programming	32
1.7 Major Contributions	33
1.8 Thesis Organization.....	35
Chapter 2. ATCoPE: Any-Time Collaborative Programming Environment	37
2.1 Introduction	37
2.2 General Design Objectives	38
2.3 ATCoPE System Architecture	41
2.3.1 The ATCoPE Server	41
2.3.2 The ATCoPE Client	42
2.4 Working Process and Major Functionalities.....	43

2.4.1	Logging into the System: User Account and Source Code Repository Management	43
2.4.2	Creating a Session: Start of Collaborative Work	44
2.4.3	Joining a Session: Dynamic Session Membership Management	45
2.4.4	Working in a Session: Features of Any-Time Collaboration.....	46
2.4.5	Leaving and Terminating a Session: Completion of Collaborative Work	49
2.4.6	Switching among Different Collaboration Modes and Sessions: Flexible Any-Time Collaboration.....	49
2.5	Summary.....	51
Chapter 3. Major Technical Issues and Solutions for Supporting ATCoPE		53
3.1	Introduction	53
3.2	Supporting Consistency Maintenance in Real-Time Collaboration Sessions	54
3.2.1	Consistency Maintenance in Real-Time Cluster Sessions.....	55
3.2.2	Consistency Maintenance in Real-Time File Sessions.....	68
3.3	Supporting Dynamic Session Membership in Real-Time Collaboration Sessions	71
3.4	Supporting Non-Real-Time Collaboration Functionalities in ATCoPE	75
3.4.1	Incorporating Non-Real-Time Collaboration Services for Non-Real-Time Collaborative Programming	75
3.4.2	Bridging Non-Real-Time Collaboration Services and Real-Time Collaboration Sessions	76
3.5	Summary.....	80
Chapter 4. DAL: Dependency-based Automatic Locking for Supporting Semantic Conflict Prevention in ATCoPE.....		83
4.1	Introduction	83
4.2	Exploring Semantic Conflicts in Real-Time Collaborative Programming Scenarios.....	86
4.2.1	Case 1: Semantic Conflict between Concurrent Editing Operations in a Self-Contained Source Code Segment	88
4.2.2	Case 2: Semantic Conflict between Concurrent Editing Operations in a Method and a Referenced Field.....	90
4.2.3	Case 3: Semantic Conflict between Concurrent Editing Operations in a Method and another Invoked Method	92
4.3	Dealing with Semantic Conflicts.....	94
4.3.1	General Conditions of Semantic Conflicts	94
4.3.2	General Approaches to Dealing with Semantic Conflicts	95

4.3.3	Dependency-based Automatic Locking (DAL).....	96
4.4	Basic Concepts of the DAL Approach.....	98
4.5	DAL Permission Check: Key of the Locking Approach and Mechanism....	100
4.6	Dependency Graph and Locking State Maintenance	103
4.6.1	General Requirements	103
4.6.2	The Explicit Maintenance Approach	104
4.6.3	The Implicit Derivation Approach.....	106
4.6.4	DAL Locking State Data Structures	108
4.7	The Basic DAL Scheme	110
4.7.1	DAL Permission Check Procedure	110
4.7.2	DAL Locking State Update Procedure	115
4.7.3	Integrated Operation Handlers for Processing Editing and Locking Operations.....	119
4.7.4	DAL Locking Release Mechanisms	120
4.8	Summary.....	122
Chapter 5. Extended DAL for Supporting Semantic Conflict Prevention in Unconstrained Real-Time Collaborative Programming		125
5.1	Introduction	125
5.2	Restrictions of the Basic DAL Scheme in Supporting Unconstrained Real-Time Collaborative Programming	127
5.2.1	Restriction in Supporting Concurrent Work on Independent Working Regions with Common Depended Regions.....	127
5.2.2	Restriction in Supporting Concurrent Editing Operations with Overlapping Locking Scopes	129
5.2.3	Restriction in Supporting Dynamic DG Editing Operations	131
5.3	Shared-Locking for Addressing Restrictions of the Basic DAL Scheme.....	135
5.3.1	Shared-Locking on Common Depended Regions	135
5.3.2	Shared-Locking for Concurrent Editing Operations with Overlapping Locking Scopes	137
5.3.3	Shared-Locking for Dynamic DG Editing Operations	139
5.3.4	Common Design Rationales for Three Types of Shared-Locking....	142
5.4	The Extended DAL Scheme	144
5.4.1	Extended Locking State Data Structures for Accommodating Shared-Locking	144
5.4.2	Correctness Criteria for Locking State Maintenance	148
5.4.3	Locking State Update under the Extended DAL Scheme	150

5.4.4	Key Procedures for the Extended DAL Scheme	157
5.4.5	Correctness Verification of the CFD Locking State Update Scheme	161
5.5	Comparison with Related Work	164
5.6	Summary	166
Chapter 6.	The ATCoEclipse Prototype System: A Proof-of-Concept of ATCoPE	169
6.1	Introduction	169
6.2	The Transparent Adaptation Approach for ATCoEclipse System Implementation	170
6.3	The ATCoEclipse System	173
6.3.1	The ATCoEclipse Server	173
6.3.2	The ATCoEclipse Client	174
6.4	User Interface Design of the ATCoEclipse Client	178
6.4.1	Major User Interfaces of the ATCoEclipse Client	178
6.4.2	The DAL-related Collaboration Workspace and Locking Awareness Feature in the ATCoEclipse User Interface	182
6.5	Preliminary Performance Feedback and Analysis	186
6.5.1	General Feedback and Analysis	187
6.5.2	Micro Benchmark Experiments	188
6.6	Summary	193
Chapter 7.	Conclusions and Future Work	195
7.1	Conclusions	195
7.1.1	Any-Time Collaborative Programming Environment (ATCoPE)	195
7.1.2	Dependency-based Automatic Locking (DAL)	197
7.1.3	The ATCoEclipse Prototype System	198
7.2	Future Work	199
7.2.1	ATCoPE Supporting Techniques	200
7.2.2	DAL Supporting Techniques	202
7.2.3	ATCoPE System Building Techniques and Usability Study	203
Bibliography		205
Publications Derived from this Research		216

List of Figures

Figure 1.1: The software development process and phases.....	3
Figure 1.2: Non-real-time collaborative programming supported by version control systems	10
Figure 1.3: Real-time collaborative programming on the contents of the same source code file	14
Figure 1.4: Initial assignment of programming tasks	25
Figure 1.5: Dynamic collaboration needs at different stages of an example collaborative programming process	27
Figure 1.6: Transitions among collaboration patterns in an example collaborative programming process	31
Figure 2.1: The ATCoPE system architecture	41
Figure 3.1: An example of the conflict among file-level editing operations.....	69
Figure 3.2: OT-based conflict resolution for file-level editing operations	70
Figure 3.3: The distributed join-protocol for dynamic session membership.....	72
Figure 3.4: Distributed leave-protocol for dynamic session membership.....	74
Figure 3.5: Incorporating non-real-time collaboration services for non-real-time collaborative programming	76
Figure 3.6: Bridging non-real-time collaboration services and real-time collaboration sessions	78
Figure 4.1: Java source code of the <i>Stack</i> implementation.....	88
Figure 4.2: Semantic conflict between concurrent editing operations in a self-contained source code segment.....	89
Figure 4.3: Semantic conflict between concurrent editing operations in a method and a referenced field	91
Figure 4.4: Semantic conflict between concurrent editing operations in a method and another invoked method	93
Figure 4.5: Basic region, dependency relationship, and dependency graph.....	99
Figure 4.6: Simple illustration of the DAL permission check.....	102
Figure 4.7: DAL locking state data structures	109
Figure 5.1: Restriction in supporting concurrent work on independent working regions with common depended regions	128
Figure 5.2: Locking operation conflict caused by concurrent editing operations	131

Figure 5.3: Dynamic DG editing operations	132
Figure 5.4: Locking operation conflict caused by dynamic DG editing operations without concurrency involved	134
Figure 5.5: Locking operation conflict caused by dynamic DG editing operations in combination with concurrency	134
Figure 5.6: Shared-locking on common depended regions	135
Figure 5.7: Shared-locking for concurrent editing operations (W-W Overlapping)	138
Figure 5.8: Shared-locking for concurrent editing operations (W-D Overlapping)	138
Figure 5.9: Shared-locking for dynamic DG editing operations (without concurrency involved).....	140
Figure 5.10: Shared-locking for dynamic DG editing operations (in combination with concurrency)	141
Figure 5.11: Three-level hierarchical data structures for accommodating shared-locking under the extended DAL scheme.....	145
Figure 5.12: An instance of the DAL locking state data structures	146
Figure 5.13: Contextualization of editing positions	152
Figure 5.14: Locking state update by partial derivation.....	154
Figure 5.15: Locking state update by full derivation	155
Figure 5.16: A comprehensive example of CFD locking state update	156
Figure 6.1: The ATCoEclipse system architecture	173
Figure 6.2: The ATCoEclipse Client system architecture and major functional modules	175
Figure 6.3: The user interface of the ATCoEclipse Client for initializing collaborative programming work	178
Figure 6.4: The major user interface of the ATCoEclipse Client	180
Figure 6.5: DAL-related collaboration workspace and locking awareness feature in ATCoEclipse Client user interface	185

List of Tables

Table 1.1: Comparisons between non-real-time and real-time collaborative programming paradigms.....	22
Table 3.1: Conflict/compatible relationships between cluster-level editing operations targeting files/directories in the same directory	61
Table 3.2: Conflict/compatible relationships between cluster-level editing operations targeting hierarchically-related file/directories ($D_1 \neq D_2$).....	61
Table 3.3: Specifications of resolution results for conflicting operations targeting files/directories in the same directory.....	62
Table 3.4: Specifications of resolution results for conflicting operations targeting hierarchically-related file/directories ($D_1 \neq D_2$)	64
Table 4.1: Utility functions for DAL permission check procedure	110
Table 4.2: Utility functions for DAL locking state update procedure	115
Table 5.1: Different types of shared-locking allowed under different conditions	150
Table 6.1: Summary of functional design objectives and technical solutions.....	186
Table 6.2: Results of performance measurement on utility functions related to source code contents and the dependency graph	190
Table 6.3: Results of performance measurement on utility functions related to locking state data structures.....	191
Table 6.4: Results of performance measurement on essential procedures	192

Abbreviations and Acronyms

2D	Two-Dimensional
3D	Three-Dimensional
API	Application Programming Interface
ATCoEclipse	Any-Time Collaborative Programming with Eclipse
ATCoPE	Any-Time Collaborative Programming Environment
CFD	Contextualization and Full Derivation
CVS	Concurrent Versions System
D-D Overlapping	Overlapping of Depended Regions
DAL	Dependency-based Automatic Locking
DG	Dependency Graph
DG-DG	Dependency Graph - Dependency Graph
DG-SC	Dependency Graph - Source Code
DRS	Depended Region Set
EP	Editing Position
FIFO	First In First Out
GCE	Generic Collaborative Engine
IDE	Integrated Development Environment
LOH	Local Operation Handler
NRTCOS	Non-Real-Time Collaboration Service
OT	Operational Transformation
ROH	Remote Operation Handler

RTCoS	Real-Time Collaboration Service
SCM	Software Configuration Management
SVN	Subversion
TA	Transparent Adaptation
W-D Overlapping	Overlapping of Working Region and Depended Region
W-W Overlapping	Overlapping of Working Regions

Abstract

Programming is the process of designing, writing, testing, debugging and maintaining the source code of software systems. Collaboration is needed in programming due to the requirements for diverse expertise and skills to solve complex problems and for producing increasingly large and complex software systems within tight schedules imposed by competitive software markets. Collaborative programming techniques and environments have been active areas of research and development in both academic and industrial worlds, and past research has invented two complementary collaborative programming paradigms, namely non-real-time collaborative programming and real-time collaborative programming, with different characteristics, applications and supporting techniques. Non-real-time collaborative programming is most suitable for large-scope and independent programming tasks involving a large number of programmers with coordinated, infrequent and manual interactions for long durations, whereas real-time collaborative programming is most suitable for small-scope and interdependent programming tasks involving a small team of programmers with ad hoc, frequent and automatic interactions for short durations. This research proposes a novel collaborative programming paradigm named *any-time collaborative programming*, which aims to seamlessly integrate both real-time and non-real-time collaborative programming to meet the complementary and dynamic collaboration needs in programming processes. Under the any-time collaborative programming paradigm, multiple programmers may work in real-time and/or non-real-time collaboration modes supported by the most suitable techniques and tools, and switch among different collaboration modes flexibly as new collaboration needs arise dynamically during the programming process. This research fo-

cuses on the architecture and enabling techniques for supporting and realizing any-time collaborative programming, and has made important contributions in three areas.

The first main contribution is the design of the architecture and major functional components of an *Any-Time Collaborative Programming Environment* (ATCoPE), which converts the abstract notion of any-time collaborative programming into concrete system architecture and components, and provides a framework for investigation and experimentation with any-time collaboration enabling techniques. We specified general design objectives, working process and functionalities of ATCoPE, and devised solutions for a range of ATCoPE issues, including seamless integration of existing non-real-time collaborative programming tools and environments with advanced real-time collaboration techniques, consistency maintenance and collaboration session management, and flexible interaction and transition among different collaboration modes and sessions by programmers.

The second main contribution is a package of novel techniques for supporting semantic conflict prevention in real-time collaborative programming under the ATCoPE framework. One key challenge in supporting real-time collaborative programming is semantic conflicts, which may occur when multiple programmers are performing concurrent and incompatible work in interdependent programming segments. Semantic conflicts may result in programming errors that are difficult and costly to detect and resolve. This research has conducted in-depth analysis on representative programming scenarios for understanding the nature and general conditions of semantic conflicts, and proposed a *Dependency-based Automatic Locking* (DAL) approach for supporting semantic conflict prevention. The novelty of the DAL approach is its capabilities of supporting automatic, responsive and fine-grained locking on interdependent programming segments and bal-

ancing conflict prevention, concurrent work and programmer convenience in real-time collaborative programming. Major technical innovations under the DAL approach include conditions and algorithms for the DAL permission check, techniques for efficient derivation and consistency maintenance of the source code dependency graph and locking states, shared-locking schemes for supporting semantic conflict prevention in unconstrained real-time collaboration, and contextualization and full derivation techniques for consistent locking state update with correctness verification.

The third main contribution is the design and implementation of the *ATCoEclipse* system, which serves as a proof-of-concept of ATCoPE and an experimental research vehicle for investigating ATCoPE system building techniques. Within the ATCoEclipse system, all major technical solutions devised in this research have been successfully implemented and validated, which provides positive confirmation of their feasibility and preliminary performance feedback. In addition, some user interface issues in supporting ATCoPE, particularly the collaboration workspace and locking awareness issue in the presence of the DAL scheme, are also studied during the system building work. Last but not least, the implementation of the ATCoEclipse system has also extended the *Transparent Adaptation* approach for building any-time collaborative programming systems by integrating existing single-user and non-real-time collaborative programming systems with novel real-time collaborative programming techniques which support both syntactic and semantic consistency maintenance features, without changing the source code of existing systems.

Chapter 1.

Introduction

1.1 Collaborative Software Development

1.1.1 Collaboration in Software Development

With the fast development of the global economy and technologies, the software industry is rapidly growing with increasing demands on large-scale and sophisticated software systems [23]. Building modern software systems has been difficult, and will continue to be challenging, because modern software systems have several inherent properties, including complexity, conformity, changeability and invisibility [22]. All these properties collectively make software development a complex and costly process, and many software projects are facing the challenges of missed schedules, blown budgets, and flawed products [103].

The success of a software project requires sophisticated collaboration among multiple software engineers involved in software development. To complete a software project under the constraints of time and budget, one critical factor is the effectiveness of the collaboration among multiple software engineers with diverse skills, knowledge and expertise [16][17][18][23]. Collaboration is at the heart of software development, and it has been reported that approximately 70% of a software engineer's time is spent on collaborative activities of software projects [81][104].

However, effective collaboration among multiple software engineers is often difficult to achieve, and it becomes even more difficult with the increase of the project size and

complexity [55]. In large-scale software projects, there are several characteristics of software development that make collaboration challenging, including scale, uncertainty, interdependence, and informal communication [23][32][55].

For medium- and large-size software projects, software engineers are often distributed at multiple locations, sometimes even inter-continently, bringing more challenges to collaborative software development [50][82]. Such phenomenon is commonly regarded as global software development (also known as distributed software development or multi-site software development), and there is a number of business reasons that have contributed to this trend in the software industry [84]. Physical proximity helps facilitate interactions among collaborators in software development [51], and distance among software engineers will certainly lead to problems in communication and coordination, as well as affecting the performance of distributed development teams [25][48][51].

1.1.2 Collaborative Activities in Software Development Phases

Software development is a process that transforms software requirements into products and solutions. As illustrated in Figure 1.1, a software development process (also known as a software development life cycle or simply a lifecycle) contains multiple phases, including (1) requirements specification, (2) design, (3) implementation, (4) testing, and (5) maintenance [76][88].

There are various kinds of collaborative activities spanning the entire software development life cycle. A wide range of techniques and tools have been developed and adopted for supporting different collaborative activities in different software development phases [81][105]:

Figure 1.1: The software development process and phases

1. In the requirements specification phase, engineers collaborate with each other and with end-users and stakeholders of the software project for discussing and negotiating the requirements of software projects, in order to establish the scope and capabilities for the software systems to be developed [62]. Representative collaborative requirements specification tools and systems include *WinWin* [19][20], *IBM Rational DOORS*¹ and *IBM Rational RequisitePro*².
2. In the design phase, system architects and software designers work collaboratively on system architectures and high-level design issues, driving convergence towards a final architecture and design for the project [47]. In the process of conducting design tasks, a critical issue is to manage the dependencies among programming arti-

¹ <http://www.ibm.com/software/awdtools/doors>

² <http://www.ibm.com/software/awdtools/reqpro>

facts and modules, by means of various design activities such as the modularization [60][90]. The modularization plays an important role in reducing and clarifying dependencies among programming tasks and isolating individual software developers' work from each other, which is directly related to achieving effective collaboration in the follow-up implementation phase [33]. Representative collaborative software design tools include *IBM Rational Rhapsody*³, *IBM Rational Rose*⁴ and *IBM Rational Software Architect*⁵, as well as other collaborative UML diagram design tools such as *Creately*⁶, *Lucidchart*⁷ and *ArchStudio*⁸.

3. In the implementation phase, multiple programmers collaborate with each other to produce the source code of the software systems, transforming software designers' ideas into actual solutions of software products. As innovative techniques for supporting collaborative programming are the focus of this research, more detailed elaboration on issues and techniques for collaborative programming will be discussed in Sections 1.2, 1.3 and 1.4.
4. In the testing phase, engineers jointly detect, identify and record errors (bugs) of the software systems being developed, and resolve errors collaboratively with the pro-

³ <http://www.ibm.com/software/awdtools/rhapsody>

⁴ <http://www.ibm.com/software/awdtools/developer/rose>

⁵ <http://www.ibm.com/software/awdtools/swarchitect>

⁶ <http://creately.com>

⁷ <http://www.lucidchart.com>

⁸ <http://www.isr.uci.edu/projects/archstudio>

grammers who are responsible for the source code in which errors are located. In this phase, multiple software engineers with different roles work together to control the quality of software projects. For facilitating collaboration in software testing, there exist sophisticated bug tracking and management tools such as *IBM Rational ClearQuest*⁹ and *IBM Rational ClearDDTS*¹⁰.

In addition to the above specialized tools and techniques that are dedicated to supporting specific collaborative activities and software development phases, there also exist general software project management tools for assisting the control of the software development life cycle, such as *Microsoft Project*¹¹, *Trac*¹² and *IBM Rational Team Concert*¹³. These tools can assist various software project management tasks, including process controlling, task scheduling, and resource assignment, as well as the recording and management of the organizational memory [2].

1.2 The Need for Collaborative Programming

Programming is a process of designing, writing, testing, debugging and maintaining the source code of software systems. While programming is mainly conducted in the implementation phase of the software development, it is a complex process involving mul-

⁹ <http://www.ibm.com/software/awdtools/clearquest>

¹⁰ <http://www.ibm.com/software/awdtools/clearddts>

¹¹ <http://www.microsoft.com/project>

¹² <http://trac.edgewall.org>

¹³ <http://www.ibm.com/software/rational/products/rtc>

multiple activities. The design task involved in programming focuses on low-level design issues and programming elements (e.g., detailed data structures and method interfaces in implementing a Java/C# class), whereas the design phase in the software development process focuses on design issues at higher-levels and larger-scopes (e.g., definitions and interface issues among high-level modules in hierarchical structures). Similarly, testing and debugging activities involved in programming focus on low-level and small-scope source code that is being developed, whereas the testing phase in the software development process focuses on higher-level and larger-scope testing and integrations among multiple modules of the software system.

Programming is not a linear process that follows a particular order of the above tasks. Practically, the design, coding, testing and debugging activities are usually interrelated and even mixed, and an individual programmer may subconsciously switch among different kinds of programming activities. For example, a programmer may start implementing and testing a Java class based on tentative design ideas with a class skeleton, and later on, as the implementation evolves, the programmer may continuously modify data structure definitions and method interfaces. The multiple activities in the programming work impose the requirements on programmers for diverse skill sets, knowledge, expertise and experiences, including specialized knowledge in the application domain, fine-grained design on interfaces and data structures, specialized algorithm design and implementation, and testing/reviewing/refining, as well as the source code documentation. However, an individual programmer is always limited in knowledge and experience, so it is often necessary to have multiple programmers with different skills to work collaboratively in a software development project.

Furthermore, collaboration among multiple programmers is needed to reduce software

development time and meet tight deadlines imposed by the highly competitive software markets. For example, it has been reported that Windows XP was produced by 1,800 developers and 2,200 testers within 3 years (from 1999 to 2001) [61][106], and the system was compiled from 45 million lines of source code [68].

Last but not least, collaboration among multiple programmers is needed to solve difficult programming problems and produce source code with higher quality. Programmers often encounter problems that are too difficult to solve by an individual programmer alone, and need other programmers (with the same or different skills) to work together in seeking suitable or better solutions based on their collective problem-solving capabilities, complementary knowledge and programming experiences, and mutual inspirations during communications and interactions.

In summary, collaboration among multiple programmers is generally needed for meeting the requirements of diverse programming skills and experiences in software development, for completing time-consuming programming tasks within tight deadlines, and for solving difficult programming problems with high quality.

The importance of collaboration in programming has also been well recognized in the software industry and education institutes. Many programming courses in computer science or software engineering have adopted group projects as course assignments in order to train students and develop their practical collaboration skills in environments that are similar to the real-world software industry [31][108].

Collaborative programming techniques and environments have been active areas of research and development in both academic and industrial worlds [43][81][104][105]. Past research and practice have invented two collaborative programming paradigms,

namely *non-real-time collaborative programming* and *real-time collaborative programming*, with different characteristics and properties, complementary needs and applications, and respective supporting techniques, which are elaborated in the following two sections respectively.

1.3 Non-Real-Time Collaborative Programming

Non-real-time collaborative programming supports a team of programmers to access a shared collection of programming artifacts (source code files and directories), conduct programming tasks individually, and merge/synchronize their changes on the programming artifacts at certain stages. During non-real-time collaborative programming, each individual programmer conducts the programming work in his/her local and private workspace with infrequent and limited interaction with other collaborating programmers during most of the development time. In general, such kind of collaborative programming needs support in the following two aspects.

Firstly, as the programming task of each individual programmer is conducted independently, the targeted programming modules of each programmer must be isolated from those of others with limited and fixed interdependencies. Generally, the isolation of programming tasks and modules can be achieved by modularization and decomposition in the software design, which are usually conducted based on thorough design considerations (e.g., compatibility, extensibility, maintainability, modularity and reusability of the software system being developed) and a number of design principles (e.g., high cohesion and low coupling, separation of concerns) that have been accumulated and validated from the past experiences of software engineering.

Secondly, such kind of collaborative programming paradigm should be realized by certain techniques, which must support concurrent and independent programming work and allow convenient integration and synchronization of parallel changes at certain stages. In practice, non-real-time collaborative programming is commonly supported by *software configuration management* (SCM) systems.

An SCM system is essentially a *version control system*, which is generic for managing any collection of files and directories that are being collaboratively edited by distributed users. A version control system also manages the changes made to the files and directories over time, and allows users to recover older versions of the files or examine the history of how they were changed. Built on top of version control systems, SCM systems are tailored to managing source code files and directories (regarded as source code trees/clusters) with additional features that are specific to software development. Although version control systems were initially proposed for keeping track of the evolution of file collections, they are nowadays widely adopted for facilitating and coordinating non-real-time collaborative programming on shared source code trees/clusters in software development, as well as supporting generic non-real-time collaborative editing work on any type of files [36].

With the support of version control systems, each collaborating programmer individually interacts with the version control repository to access and exchange (either downloading or uploading) the source code copy, and to integrate/synchronize the changes by multiple programmers [101][102]. As illustrated in Figure 1.2, modern version control systems commonly support the following working process for non-real-time collaborative programming, based on a *copy-modify-merge* model [75]:

Figure 1.2: Non-real-time collaborative programming supported by version control systems

1. The programmer *checks out* a cluster of source code files and directories (as a source code tree) that are related to the programming task assigned to this individual programmer. After executing the *check-out* version control command, a copy of the specified cluster of source code is downloaded from the version control repository to the programmer's local workspace.
2. The programmer *modifies* the source code copy in the local workspace to complete the programming work. In general, the following changes can be performed on the source code copy:
 - a) To *create/delete/rename* a source code file or sub-directory in any existing directory; and
 - b) To edit the contents of any source code file.
3. The programmer may *merge* the current source code copy in the local workspace

with the latest copy in the version control repository in two ways: (1) by issuing an *update*¹⁴ command, other (remote) programmers' changes available at the repository are downloaded to the local workspace and merged with the local source code copy; and (2) by issuing a *commit*¹⁵ version control command, the current version of the local source code copy is uploaded to the version control repository¹⁶, which makes the local programmer's changes visible and accessible to others.

Modern version control systems commonly permit multiple collaborators to edit the same artifact concurrently (i.e., parallel changes on a common base version). During the merge process triggered by *update* and *commit* commands, the version control system uses internal mechanisms and tools to automatically detect and resolve *conflicts* among *concurrent* changes by multiple programmers [67]. Textual merging and conflict detection tools are available in most modern version control systems [24][30][74]. If the system is unable to reconcile conflicting changes automatically, it leaves the *conflict resolution* to programmers. For example, if concurrent changes are made to the contents of the same source code file by two collaborating programmers, conflicts may be detected and reported.

By means of version control systems, programmers can merge and synchronize their

¹⁴ In some version control systems, this command is named as *rebase*.

¹⁵ In some version control systems, this command is named as *check-in*.

¹⁶ While the latest source code copy in the version control repository is updated, old versions are also maintained automatically by the version control system for revision control purpose (e.g., for supporting recovery at a later moment).

updated source code copies at certain stages of the collaborative programming work. Such kind of collaborative programming is regarded as *non-real-time collaborative programming* because changes performed by individual programmers are not immediately propagated and merged with others' copies. The local copy is kept *private* until this programmer manually *commits* local changes into the *public* repository, and in addition, other programmers have to explicitly perform *update* commands to incorporate the latest committed versions in the repository into their local copies. An individual programmer can edit, compile, test and debug the local source code continuously without being interrupted by parallel changes performed by others. Collaborating programmers can also make use of the revision management functionalities provided by the version control system, which are also needed in most software development projects.

Non-real-time collaborative programming is also regarded as *loosely-coupled* collaboration as it requires infrequent interaction among programmers during most of the time and limited integrations of source code copies at pre-scheduled stages only (e.g., once in a day). For example, in large-scale software projects, a regular collaboration setting may require programmers to commit their latest source code copies into the repository after the daily work before leaving, and the central system will consequently perform nightly build procedures in the evening and generate an integrated set as a basis for the programming work in the next day.

The non-real-time collaborative programming paradigm has been widely adopted in the software industry, which can be realized by a variety of popular and sophisticated ver-

sion control systems such as *Concurrent Versions System (CVS)*¹⁷ [11][14], *Subversion (SVN)*¹⁸ [29], *Perforce*¹⁹, *Git*²⁰ [27][59] and *IBM Rational ClearCase*²¹ [9].

1.4 Real-Time Collaborative Programming

Different from non-real-time collaborative programming, real-time collaborative programming supports a team of programmers to work on shared programming artifacts (source code files and directories) concurrently in a closely-coupled fashion, where changes performed by multiple programmers are instantly propagated and merged [73][85][87]. In addition, another important characteristic of real-time collaborative programming is that the real-time operation propagation and integration are achieved automatically by the underlying system, without requiring programmers to perform manual operations (such as the *update* and *commit* version control commands) as they do in non-real-time collaborative programming.

In real-time collaborative programming, multiple programmers can concurrently access and edit any part of the shared source code copy at the same time, and they can even edit the contents of the same source code file concurrently, as illustrated in Figure 1.3:

1. One programmer's editing operations on creating, deleting and/or renaming files

¹⁷ <http://www.nongnu.org/cvs>

¹⁸ <http://subversion.apache.org>

¹⁹ <http://www.perforce.com>

²⁰ <http://git-scm.com>

²¹ <http://www.ibm.com/software/awdtools/clearcase>

and/or directories in the shared source code copy are immediately propagated and executed, thus immediately made visible, at all remote collaborating sites, as if the programmer is performing the same editing operations at all collaborating sites.

2. Multiple programmers are also allowed to work jointly inside the same source code file at the same time, and their editing operations performed on the contents of the same source code file are instantly propagated to others for real-time notification and merging, as if a group of collaborating programmers are sitting together and jointly editing the same source code file.

Figure 1.3: Real-time collaborative programming on the contents of the same source code file

Such kind of collaborative programming facilitates fine-grained and instant interaction among collaborating programmers and automatic merging/synchronization among con-

current changes performed by programmers. As real-time collaborative programming is an emerging collaborative programming paradigm with novel supporting techniques, we present more detailed analysis on the needs for real-time collaborative programming.

1. Multiple programmers may need to work jointly on the same self-contained programming module. Such collaboration need may arise under various circumstances, including but not limited to the following:

- a) The size (workload) of a self-contained programming module may be too big for a single programmer to complete under the given time constraint, but it is difficult or impossible to further decompose this module into smaller modules due to the interdependencies among various components/elements inside it. For example, in the *Abstract Window Toolkit* (AWT) package²² of the *Java Platform Standard Edition* (Java SE)²³, a single class named *java.awt.Component* contains approximately 4500 lines of source code (as implemented in JDK 7 build b147)²⁴. Under such circumstances, multiple programmers may be assigned to work on the same module in a closely-coupled fashion, which helps to solve challenging problems in programming task decomposition and scheduling.

- b) It is oftentimes difficult to estimate the complexity of programming tasks and

²² <http://docs.oracle.com/javase/7/docs/technotes/guides/awt/index.html>

²³ <http://www.oracle.com/us/technologies/java/standard-edition/overview/index.html>

²⁴ <http://download.java.net/openjdk/jdk7>

predict what problems may occur in advance. When one programmer who is responsible for a programming module encounters unexpected difficult problems and is not able to solve them individually, the project leader may spontaneously assign additional programmers to help solve those problems with the primary programmer in a closely-coupled manner, thus removing potential development bottlenecks in the software project.

- c) Pair programming²⁵ is one way of practicing extreme programming in agile software development²⁶ and has the promise of producing high quality source code due to peer review and closely-coupled collaboration. Real-time collaborative programming naturally supports pair programming: two programmers can conduct a kind of virtual pair programming²⁷ where they are geographically dispersed and may work on the same or different parts of the same programming task flexibly [42][83]. In addition, such kind of pair programming

²⁵ In pair programming, two programmers sit side-by-side at the same workstation and collaboratively work on the same task (e.g., design, coding, testing) [28][107]. One programmer (with the role of *driver*) edits the source code by manipulating the input devices (e.g., keyboard and mouse) while the other programmer (with the role of *observer*) provides assistance in various aspects (e.g., monitoring, reviewing, problem-solving, algorithm design).

²⁶ Agile software development is an emerging method for software projects to effectively adapt to rapid changes of users' requirements during the software development life cycle, to embrace the changes, to become more productive, and to incorporate changes quickly into an evolving software product [12][13][26][58][69].

²⁷ Virtual pair programming is also known as remote pair programming or distributed pair programming.

can also be used to support software maintenance and support. For example, one engineer at the development center of a software corporation may provide remote assistance to a supporting engineer at the customer's site in collaboratively diagnosing the system and fixing problems by means of real-time collaborative programming.

2. Multiple programmers may need to jointly work on the same set of interdependent programming modules. Such collaboration need may arise under various circumstances, including but not limited to the following:

a) Multiple programming modules may have tentatively defined (thus unclear and changeable) interfaces and functionalities at initial stages of programming processes. Such situations often occur when there exist multiple iterations of design, coding, testing and debugging stages in a programming process, particularly under the agile software development method. Programming tasks on these modules are often interrelated, and the change-dependencies among these modules are much stronger than those among well-isolated modules. Under such circumstances, it is difficult or even infeasible for collaborating programmers involved in those modules to work independently, and it is beneficial for them to work in a closely-coupled real-time collaborative programming fashion to solve those interface issues and interdependent problems across multiple modules.

b) The need for multiple programmers to work on multiple programming modules in a closely-coupled fashion may also occur when programmers are conducting cross-module debugging, testing and integration, which are generally required

in all software development processes, particularly in agile software development processes.

Another important application domain of real-time collaborative programming is in education institutes, where practical sessions of programming courses can be conducted by means of real-time collaborative programming tools and systems as the educational groupware for students to follow the instructor by viewing and writing source code files together, and to collaboratively practice programming skills [63][64][65][110]. Because real-time collaborative programming is still an emerging paradigm, potential needs for real-time collaborative programming in practical software development are well beyond what we have already known.

Some early empirical studies have indicated that real-time collaborative programming can provide a range of benefits, such as reducing programming errors, creating better design and source code, accelerating the progress of problem-solving, making programmers enjoy the work more, and thus improving both productivity and quality of software projects [28][41][73][87][107][109]. These benefits have attracted rising interests from both the software industry and research communities, and numerous real-time collaborative programming tools and environments with diverse functionalities have been built in recent years. For example, there exist several popular tools for supporting virtual pair programming, such as *Saros*²⁸ [80], *Gobby*²⁹, *Sangam*³⁰ [53],

²⁸ <http://www.saros-project.org>

²⁹ <http://gobby.0x539.de>

³⁰ <http://sangam.sourceforge.net>

*XPairtise*³¹ and *PEP*³², which provide flexible collaboration features in complementing conventional pair programming supporting tools such as the *Virtual Network Computing* (VNC) [79] which commonly adopt the shared window techniques. Besides these virtual pair programming supporting tools, there is also a variety of other real-time collaborative programming systems, including research prototypes such as *RECIPE* [85][87], *Ripple* [21], *GREWTool* [52], *GHT* [57], *Moomba* [78] and *Collabode* [45], as well as commercial products such as *SubEthaEdit*³³, *beWeeVee*³⁴, *Cola*³⁵ and *VS Anywhere*³⁶.

For supporting the abovementioned functionalities and features (e.g., real-time operation propagation and integration) for real-time collaborative programming, one of the key enabling techniques is the real-time collaborative editing technique [94][95], which supports multiple geographically distributed collaborators to concurrently edit the contents of the same shared file (e.g., plain text document, rich text document, graphical document) at any time (achieving *unconstrained collaboration*), see their own updates instantly (achieving *responsive interaction*), and incorporate other collaborators' changes quickly (achieving *real-time notification and merging*). Detailed technical issues and solutions for real-time collaborative programming will be discussed in Section 3.2.

³¹ <http://xpairtise.sourceforge.net>

³² <http://pep-pp.sourceforge.net>

³³ <http://www.codingmonkeys.de/subethaedit>

³⁴ <http://www.beweevee.com>

³⁵ <http://live.eclipse.org/node/543>

³⁶ <http://www.vsanywhere.com>

1.5 Characteristics, Techniques, and Dynamic Needs for Non-Real-Time and Real-Time Collaborative Programming

1.5.1 Characteristics of Non-Real-Time and Real-Time Collaborative Programming

Based on introductions and analysis on non-real-time collaborative programming and real-time collaborative programming in Sections 1.3 and 1.4, major characteristics of the two collaborative programming paradigms can be compared as follows.

Firstly, non-real-time and real-time collaborative programming are needed for supporting programming tasks with different dependency relationships: non-real-time collaborative programming is most suitable for supporting independent programming tasks, whereas real-time collaborative programming is most suitable for supporting interdependent programming tasks.

Secondly, non-real-time collaborative programming usually involves a large number of programmers working on large-scope programming tasks for relatively long durations, whereas real-time collaborative programming usually involves a small team of programmers working on small-scope programming tasks for relatively short durations.

Thirdly, the needs for non-real-time collaborative programming can be anticipated, and the collaborative work can be planned and pre-scheduled accordingly in software projects. In contrast, the needs for real-time collaborative programming may not be anticipated in the initial project planning, but may arise dynamically in the programming process. Among various cases in which real-time collaborative programming is needed as

presented in Section 1.4, some of them may be pre-scheduled (e.g., the project manager may intentionally plan a pair-programming session or a closely-coupled real-time collaboration session for multiple programmers to solve the interdependency issues among modules in an agile software development process), while most of them may start spontaneously in response to dynamic needs for closely-coupled collaboration (e.g., assigning additional programmers to assist an individual programmer in solving unpredicted problems that are detected during the programming process).

Finally, in non-real-time collaborative programming, programmers do not need to interact and communicate with each other frequently, due to the independency of their isolated programming tasks. Furthermore, interactions among non-real-time collaborating programmers are often coordinated and controlled manually by programmers. For example, they need to explicitly issue *update* and *commit* commands to synchronize their work and resolve conflicts manually by means of the version control system. In contrast, real-time collaborating programmers often need to interact and communicate frequently due to the interdependencies among their non-isolated programming tasks, and such interactions are often ad hoc and facilitated automatically by the underlying real-time collaborative programming techniques [86].

To summarize, some major differences between non-real-time and real-time collaborative programming paradigms are highlighted in Table 1.1.

Table 1.1: Comparisons between non-real-time and real-time collaborative programming paradigms

Aspect of Comparison	Non-Real-Time Collaborative Programming	Real-Time Collaborative Programming
Relationships among Programming Tasks/Modules	<i>Independent</i>	<i>Interdependent</i>
Scope of Programming Tasks	<i>Large</i>	<i>Small</i>
Size of Programming Team	<i>Large</i>	<i>Small</i>
Collaboration Duration	<i>Long</i>	<i>Short</i>
Collaboration Scheduling	<i>Pre-scheduled</i>	<i>Spontaneous</i>
Interaction Style	<i>Coordinated</i>	<i>Ad Hoc</i>
Interaction Frequency	<i>Infrequent</i>	<i>Frequent</i>
Interaction Control	<i>Manual</i>	<i>Automatic</i>
Conflict Resolution	<i>Manual</i>	<i>Automatic</i>

1.5.2 Complementary Techniques for Supporting Non-Real-Time and Real-Time Collaborative Programming

In recognizing the needs for both non-real-time and real-time collaborative programming, various techniques and tools have been invented to support each of them respectively. Non-real-time collaborative programming has been commonly supported by techniques and tools (such as version control systems) that are suitable for large-size team work, convenient for programmers to work independently and synchronize their work in coordinated manners, and efficient in supporting coarse-grained (e.g., file-level) communications and revision control. In contrast, real-time collaborative programming has been supported by techniques and tools that are suitable for small-size team work,

convenient for programmers to interact and communicate frequently in ad hoc manners, and efficient in supporting frequent and fine-grained (e.g., operation-level) communications, including early application-sharing systems (e.g., XTV [1], SharedX [44], VNC [79], NetMeeting³⁷ [91]) and some emerging real-time collaborative systems as introduced in Section 1.4. It is important to recognize that the two collaborative programming paradigms require different and complementary supporting techniques that cannot substitute each other.

Non-real-time collaboration supporting techniques may not be suitable for supporting real-time collaborative programming. Suppose a group of collaborating programmers need to work in a closely-coupled real-time collaborative programming session but they are supported by conventional non-real-time collaboration tools and environments, such as version control systems like SVN. Under such technical constraints, programmers have to manually and frequently perform version control operations (e.g., issuing *update* and *commit* commands) for propagating incremental updates and synchronizing latest changes with each other, which is clearly inconvenient for programmers as they are frequently diverted from their major programming work to take care of communication issues. Such way of communication and synchronization is also very inefficient because it invokes heavy-duty (file-level) communication mechanisms for supporting frequent and minor updates, which not only causes high communication overheads, but also creates unnecessary versions inside the version control repository. Furthermore, when multiple programmers are working in the same source code file and updating ad-

³⁷ <http://www.microsoft.com/download/details.aspx?id=23745>

jacent source code segments, they have to be involved in manual conflict resolution frequently due to the limitation of the conflict resolution mechanisms in conventional version control systems [24][67][74].

Similarly, real-time collaboration supporting techniques may not be suitable for supporting non-real-time collaborative programming either. If a large group of collaborating programmers need to work under a loosely-coupled non-real-time collaborative programming paradigm but supported by real-time collaboration tools and techniques, updates performed by each programmer will be instantly and automatically propagated to a large number of collaborating programmers, but most of them may not be interested in seeing those updates due to the independency of their programming tasks. Such frequent and automatic synchronization is not only inefficient in terms of unnecessary communication costs, but also violates an important requirement of non-real-time collaborative programming - *workspace isolation*: collaborating programmers should be working in their private workspaces, which are isolated in the sense that the updates performed in private workspaces are not visible to other programmers unless they manually perform communication and synchronization operations (e.g., issuing *update* and *commit* commands in version control systems). Such workspace isolation is necessary for supporting independent compilation, testing and debugging, and commonly enforced by non-real-time collaborative programming environments. However, frequent and automatic operation propagation enabled by real-time collaborative programming techniques clearly violates the requirement on workspace isolation. Last but not least, real-time collaborative programming tools and techniques do not accommodate revision management functionalities, and therefore they cannot meet the needs for source code versioning, which is commonly required in non-real-time collaboration.

1.5.3 Dynamic Collaboration Needs at Different Stages of the Programming Process

Non-real-time and real-time collaborative programming paradigms are needed not only in different programming processes under different software development methods, but also at different stages of the same programming process. In the following, we will present an example collaborative programming process to demonstrate the dynamic collaboration needs at different stages of the programming process.

Consider a team of five programmers collaboratively working for a software project. Initially, the software system to be developed may be tentatively divided into four modules. Based on the estimated complexity and workload of each module and the expertise and experiences of the available programmers, the four modules are assigned to the five programmers as shown in Figure 1.4.

Figure 1.4: Initial assignment of programming tasks

Module 1 is assigned to P_1 ; Module 2 is assigned to P_2 ; Module 3 is assigned to P_3 ; and Module 4 is assigned to two programmers P_4 and P_5 due to the estimated high complexity and workload of Module 4, the complementary skill sets and expertise possessed by P_4 and P_5 that are required for implementing Module 4, and/or the estimated availability and

productivity of P_4 and P_5 , etc.

In the above task assignment, programmers P_1 , P_2 and P_3 will be working on Module 1, Module 2 and Module 3 independently during most of the time and coordinating their programming tasks infrequently, in a loosely-coupled non-real-time collaboration fashion; while programmers P_4 and P_5 will be working on the same Module 4 in a closely-coupled real-time collaboration fashion. However, the needs for non-real-time and/or real-time collaborative programming may arise and vary from time to time at various stages during the programming process as illustrated in Figure 1.5.

At the beginning of the programming process (Stage 1), the interfaces, functionalities and semantics of the programming modules may not be clearly specified (e.g., in an agile software development fashion). Suppose that the interfaces and interdependencies between Module 1 and Module 2 are initially unclear, and the programming work on Module 1 and Module 2 are thus highly interrelated. P_1 and P_2 need to work in a closely-coupled real-time collaboration fashion to jointly resolve interface and functional issues between the two modules. Under real-time collaborative programming, P_1 and P_2 can freely interact with each other and they are aware of what the other is doing at all times. For the same reason, P_3 needs to work closely with P_4 and P_5 to resolve interface and functional issues between Module 3 and Module 4 in a real-time collaboration fashion at this stage. In addition, one group of real-time collaborating programmers (P_1 and P_2) also needs to coordinate and synchronize their work infrequently with another group of real-time collaborating programmers (P_3 , P_4 and P_5) in a loosely-coupled non-real-time collaboration fashion.

Figure 1.5: Dynamic collaboration needs at different stages of an example collaborative programming process

At the later Stage 2, the interface and functional issues between Module 1 and Module 2 have been jointly resolved by P_1 and P_2 , and the two modules now become isolated from each other. Consequently, P_1 and P_2 do not need to frequently interact with each other anymore, and they can conduct programming work independently in a loosely-coupled non-real-time collaboration fashion. Similarly, after the boundaries and interfaces between Module 3 and Module 4 have been clearly defined, P_3 can proceed to work independently on Module 3, whereas P_4 and P_5 can continue to work in the same Module 4 in a real-time collaboration fashion. At this stage, the three individual programmers (P_1 , P_2 and P_3) may need to coordinate with each other and also with the group of real-time collaborating programmers (P_4 and P_5) in a loosely-coupled non-real-time fashion.

Later on at Stage 3, P_2 and P_3 may discover some emerging problems that are related to both Module 2 and Module 3, and they need closely-coupled interaction for coordinating their changes in both modules. Under such circumstances, they need to spontaneously start a real-time collaborative programming session to jointly solve the identified problems. By means of the closely-coupled real-time collaboration, they are able to consider the interrelationships between Module 2 and Module 3 together, avoid incompatible changes, and collectively resolve identified problems. Similarly, P_1 may also encounter some unexpected problems in Module 1, which are too difficult for him to solve alone due to his limited knowledge and experience. Therefore, P_1 may request immediate assistance from P_4 , who has the expertise to solve the problems. Without terminating his real-time collaborative work with P_5 on Module 4, P_4 may instantly join P_1 in working on Module 1 and help solve the problems in a real-time collaboration fashion. Moreover, P_4 may also flexibly switch between the two real-time collaboration sessions for contributing to both Module 1 and Module 4. In general, such ad hoc collaboration needs are unpredictable,

which may arise spontaneously at any stage of the programming process and require technical support for convenient and flexible transitions among non-real-time and real-time collaboration modes, as well as the smooth transitions among different real-time collaboration sessions. At this stage, five programmers are involved in three different real-time collaborating groups and they also need to coordinate their group work infrequently in a non-real-time collaboration fashion.

At the final Stage 4, the programming process enters the stage for cross-module testing, debugging and integration, which require all collaborating programmers to work together in a closely-coupled fashion because problems detected at this stage may involve multiple programming modules. Collaborating programmers need to jointly detect and fix problems, coordinate their changes in different modules, and collectively integrate all modules to make a complete system.

For the convenience of discussion, we have illustrated the dynamic collaboration needs in an example collaborative programming process with a specific sequence of stages. However, each of the stages actually represents a distinctive collaborative programming pattern, and the applications of these collaboration patterns may not be sequential.

1. Stage 1 represents a collaboration pattern (namely Pattern 1) where there exist multiple real-time collaborating groups (sessions), and the interactions among these real-time collaborating groups are conducted in a non-real-time collaboration fashion.
2. Stage 2 represents a collaboration pattern (namely Pattern 2) where both individual programmers and real-time collaborating groups co-exist concurrently, and the interactions among individual programmers and real-time collaborating groups are conducted in a non-real-time collaboration fashion. Particularly, if Module 4 is also

developed by an individual programmer at Stage 2, it would represent a common collaboration pattern where a group of individual programmers are conducting the programming work in a conventional non-real-time collaboration fashion.

3. The collaboration pattern (namely Pattern 3) represented by Stage 3 is similar to the pattern represented by Stage 1, but with a special characteristic: one programmer is participating in multiple real-time collaborating groups at the same time and flexibly switching among them from time to time. In general, a programmer may be involved in multiple programming sessions, which may be either non-real-time (single-user) or real-time collaborative programming sessions.
4. Stage 4 represents a collaboration pattern (namely Pattern 4) where all programmers in a development team are working in the same real-time collaborating group (one real-time collaboration session) in a closely-coupled fashion.

Transitions among collaboration patterns may vary in different software projects, depending on the collaboration needs. A collaboration pattern transition graph for an example collaborative programming process is illustrated in Figure 1.6.

In this example, the conventional non-real-time collaboration Pattern 2 serves as a common collaboration pattern, from which collaborating programmers may switch to various combinations of real-time and non-real-time collaboration patterns, in either a pre-scheduled or a spontaneous fashion:

1. Pattern 1 is entered when the need arises for resolving interface and other interrelated issues among multiple modules. After the problems are solved, programmers return to Pattern 2 for independent work.

2. Pattern 3 is entered whenever a programmer spontaneously requests immediate assistance from another programmer or discovers any unexpected problem across multiple modules. After the problem is solved in a closely-coupled fashion, programmers return to Pattern 2.
3. Pattern 4 is entered when the need arises for cross-module debugging and integration. Once the integration work is completed, programmers return to Pattern 2.

Transitions among these collaboration patterns may occur multiple times during an iteration of design, coding, testing, debugging and integration, and multiple iterations may form the entire programming process.

Figure 1.6: Transitions among collaboration patterns in an example collaborative programming process

1.6 Any-Time Collaborative Programming

In recognizing the complementary roles of non-real-time and real-time collaborative programming paradigms and the dynamic collaboration needs in the programming process as illustrated in Figure 1.5, we propose a novel collaborative programming paradigm named *any-time collaborative programming*, which seamlessly integrates both non-real-time and real-time collaborative programming. Under the any-time collaborative programming paradigm, multiple programmers may work in non-real-time and/or real-time collaboration modes supported by the most suitable techniques and tools, and flexibly switch among different collaboration modes as new collaboration needs arise dynamically during the programming process.

The any-time collaborative programming paradigm is generally compatible with any type of software development method, model or process, as it focuses on the collaboration patterns during the programming process, which is a generic issue in any kind of software development. In practice, any-time collaborative programming is suitable and beneficial to any software development method/process, and such emerging collaboration paradigm may be particularly preferable with some emerging software development methods/processes such as the agile software development, because they commonly require a variety of collaboration patterns and usually promote frequent communications at certain stages of the development iteration.

To realize the any-time collaborative programming paradigm, this research has contributed an *Any-Time Collaborative Programming Environment* (ATCoPE) which converts the abstract notion of any-time collaborative programming into concrete system architectures and functionalities, and a systematic package of enabling techniques for sup-

porting any-time collaborative programming under the framework of ATCoPE. In addition, a research prototype system named *ATCoEclipse* has been designed and implemented as a proof-of-concept of ATCoPE, which serves as an experimental research vehicle for exploring research areas and issues, experimenting with technical approaches and solutions, and validating and evaluating research results.

1.7 Major Contributions

Major contributions of this thesis are summarized as follows:

1. **An *Any-Time Collaborative Programming Environment* (ATCoPE)**, which converts the abstract notion of any-time collaborative programming into concrete system architectures and functionalities. We proposed a set of general design objectives, and specified the working process and functionalities of ATCoPE from the end-users' perspective. Moreover, we devised solutions for a range of major issues in supporting ATCoPE, including the seamless integration of existing non-real-time collaborative programming tools and environments with advanced real-time collaboration techniques, the consistency maintenance of the shared source code, the dynamic session membership management in real-time collaboration sessions, and the flexible interaction and transition among different collaboration modes and sessions. The ATCoPE architecture and techniques are unique in addressing the particular needs for any-time collaborative programming, which are important contributions to collaborative software development technologies [40]. This package of contributions will be presented in Chapter 2 and Chapter 3.
2. **A package of novel techniques for supporting semantic conflict prevention in**

real-time collaborative programming under the ATCoPE framework. Based on in-depth analysis on representative programming scenarios, we derived the general conditions of semantic conflicts and proposed a novel *Dependency-based Automatic Locking* (DAL) approach for supporting semantic conflict prevention. The DAL approach is capable of supporting automatic, responsive and fine-grained locking and balancing conflict prevention, concurrent work and programmer convenience in real-time collaborative programming. We contributed a package of novel techniques and solutions for supporting the DAL approach, including conditions and algorithms for the DAL permission check, an implicit derivation approach for efficient consistency maintenance of the source code dependency graph and locking states, the shared-locking schemes for supporting semantic conflict prevention in unconstrained real-time collaborative programming, and a contextualization and full derivation scheme and techniques for achieving consistent and correct locking state update under the extended DAL scheme with correctness verification [37][39]. This package of contributions will be presented in Chapter 4 and Chapter 5.

3. **The design and implementation of the ATCoEclipse system.** All major technical solutions devised in this research have been successfully implemented and validated in the ATCoEclipse prototype system, which provides positive confirmation of their feasibility and preliminary performance feedback. In addition, some novel user interface techniques, particularly the DAL-related collaboration workspace and locking awareness feature, are also contributed. In addition, the implementation of the ATCoEclipse system has extended the *Transparent Adaptation* approach for building any-time collaborative programming systems by integrating existing single-user and non-real-time collaborative programming systems with novel real-time

collaborative programming techniques which support both syntactic and semantic consistency maintenance features, without changing the source code of existing systems [38][40]. This package of contributions will be presented in Chapter 6.

1.8 Thesis Organization

This chapter has presented the background and motivation of this research, and the rest of the thesis is organized as follows:

1. In Chapter 2, we propose the general design objectives, design the ATCoPE system architecture, and specify the working process and major functionalities of ATCoPE.
2. In Chapter 3, we discuss major technical issues and devise solutions in supporting ATCoPE, including the consistency maintenance of source code and the dynamic session membership management in real-time collaboration sessions, and providing non-real-time collaboration functionalities in both non-real-time and real-time collaboration sessions.
3. In Chapter 4, we analyze a set of programming scenarios to derive the general conditions of semantic conflicts, propose the DAL approach for supporting semantic conflict prevention, and devise major techniques for supporting the DAL approach, including the permission check, dependency graph and locking state maintenance, and the basic DAL scheme for realizing all key DAL procedures and mechanisms.
4. In Chapter 5, we present shared-locking schemes for supporting semantic conflict prevention in unconstrained real-time collaborative programming where programmers are allowed to concurrently issue editing operations which may dynamically

change the source code structure. In addition, we present a package of techniques for realizing the extended DAL scheme, including revised DAL permission check conditions, extended locking state data structures for supporting shared-locking, and the contextualization and full derivation locking state update scheme with correctness verification.

5. In Chapter 6, we present the design and implementation of the ATCoEclipse system, including the design of major user interfaces of the ATCoEclipse Client with the novel DAL-related collaboration workspace and locking awareness feature. We also present preliminary performance feedback and analysis, including a set of micro benchmark experiments.
6. In Chapter 7, we conclude this research by highlighting major contributions and discussing several research directions and issues identified for the future work.

Chapter 2.

ATCoPE: Any-Time Collaborative Programming Environment

2.1 Introduction

In Chapter 1, we presented and discussed the complementary and dynamic needs for both non-real-time and real-time collaborative programming, and motivated the any-time collaborative programming paradigm to seamlessly integrate non-real-time and real-time collaborative programming. In this chapter, we present the design of the system architecture and major functionalities of a novel *Any-Time Collaborative Programming Environment* (ATCoPE), which realizes the abstract notion of any-time collaborative programming paradigm, and provides a framework for investigation and experimentation with any-time collaboration enabling techniques.

The rest of this chapter is organized as follows. In Section 2.2, a set of general design objectives and rationales for ATCoPE are proposed. In Section 2.3, the ATCoPE system architecture and functional components, which are aimed at meeting those design objectives, are designed and described. In Section 2.4, the working process and major functionalities of ATCoPE are specified from the end-users' perspective, including how collaborating programmers interact with the ATCoPE system step by step during the working process, the semantics of the operations performed, and major functionalities and features supported by the ATCoPE system.

2.2 General Design Objectives

In this section, we propose four general design objectives for ATCoPE and discuss the rationales behind them. These design objectives will be used as the general guidance for ATCoPE technical research and system implementation.

1. Capability and Compatibility in Supporting Single-User and Non-Real-Time Collaborative Programming

ATCoPE will provide rich conventional functionalities for single-user and non-real-time collaborative programming, and preserve the compatibility with existing non-real-time collaborative programming tools and environments in terms of user interfaces, functionalities and features, working processes, and operation semantics.

In terms of the single-user programming work, the ATCoPE end-users will be able to conduct programming tasks (e.g., creating source code files, coding, debugging) in the same way as they do in existing single-user programming environments such as Microsoft Visual Studio³⁸ and Eclipse³⁹. In terms of non-real-time collaboration, programmers will be able to continue using version control commands (e.g., *commit*, *update*) to interact with other non-real-time collaborating programmers in the same way as using conventional non-real-time collaboration supporting tools such as SVN.

With this design objective, ATCoPE will support end-users to conduct single-user and non-real-time collaborative programming with the same skill sets, knowledge and expe-

³⁸ <http://www.microsoft.com/visualstudio>

³⁹ <http://www.eclipse.org>

riences as before while enjoying additional collaboration capabilities including real-time and any-time collaboration.

2. Capability of Supporting Advanced Real-Time Collaborative Programming

In addition to single-user and non-real-time collaborative programming, ATCoPE will support additional functionalities for real-time collaborative programming in the following three aspects:

Firstly, ATCoPE will support multiple programmers to freely and concurrently work in the same collection of source code files and directories for the same project at the same time. Programmers will be able to perform any editing operation to the shared source code, and see other collaborators' updates in real-time. In the face of concurrent updates performed by collaborating programmers, ATCoPE will automatically resolve conflicts and maintain the consistency of the shared source code at all times.

Secondly, ATCoPE will provide innovative functionalities to assist collaborating programmers in dealing with incompatibilities and conflicts among concurrent programming work. Incompatibilities and conflicts are more likely to occur in closely-coupled real-time collaboration than in loosely-coupled non-real-time collaboration as real-time collaborative programming tasks usually have stronger interdependencies.

Thirdly, ATCoPE will provide novel collaboration awareness features to programmers. Collaboration awareness is an important factor in achieving effective collaborative programming work because real-time collaboration requires frequent communication and coordination among collaborating programmers. Collaboration awareness features will also assist programmers in dealing with incompatibility and conflict issues.

3. Capability of Supporting Any-Time Collaborative Programming

ATCoPE will support not only simultaneous non-real-time and real-time collaboration sessions for different projects, but also any-time collaboration sessions for the same project. An any-time collaboration session comes into existence when there exist both non-real-time and real-time collaboration sessions for the same project (as illustrated by the example in Section 1.5.3). In an any-time collaboration session, programmers may work individually, collaborate with each other in a loosely-coupled non-real-time fashion, and/or work in a closely-coupled real-time fashion. ATCoPE will be able to support multiple non-real-time and real-time collaboration sessions for the same or different projects, allow programmers to switch among different collaboration modes and sessions conveniently according to their collaboration needs, and support real-time collaborating programmers to use non-real-time collaboration functionalities (e.g., *update* and *commit*) and collectively resolve non-real-time collaboration conflicts (i.e., conflicts with other real-time collaborating groups or single-user programmers) in a real-time collaboration fashion.

4. High Performance and Scalability

ATCoPE will provide end-users with high local responsiveness (as responsive as single-user programming tools and environments), fast remote notification for real-time collaborating programmers, and efficient usage of client system resources, server system resources and communication bandwidth.

ATCoPE will be able to accommodate a large number of end-users and collaboration sessions, and maintain good system performance as the increase of the number of active collaborating programmers and the number of simultaneous active collaboration ses-

sions for large-scale software projects.

2.3 ATCoPE System Architecture

Guided by the design objectives in Section 2.2, we propose the ATCoPE system architecture as shown in Figure 2.1. The ATCoPE system consists of one *ATCoPE Server* and multiple *ATCoPE Clients* connected by communication networks.

Figure 2.1: The ATCoPE system architecture

2.3.1 The ATCoPE Server

The ATCoPE Server consists of a *Real-Time Collaboration Service* (RTCoS) component, a *Non-Real-Time Collaboration Service* (NRTCoS) component, and a uniform *ATCoPE Server Interface* for supporting any-time collaboration.

The NRTCoS component is responsible for user account management, source code repository management, and source code versioning management, which are common

functionalities required for non-real-time collaboration. To meet Design Objective 1, the NRTCoS component supports these conventional non-real-time collaboration functionalities by incorporating existing version control systems.

The RTCoS component is responsible for supporting advanced real-time collaboration functionalities (to achieve Design Objective 2), including real-time collaboration session management, source code cache management (to be discussed in Section 3.4.2), and real-time group membership management. The RTCoS component takes advantage of the NRTCoS component for reusing user account management and source code repository management functionalities, as well as other non-real-time collaboration functionalities for supporting loosely-coupled collaboration among real-time collaborating groups and single-user programmers (to achieve Design Objective 3).

2.3.2 The ATCoPE Client

The ATCoPE Client provides an integrated development environment (IDE) with comprehensive conventional programming functionalities, such as coding, compilation and debugging, as well as various supporting tools and interfaces such as class browser and source code editor. The IDE is also integrated with version control functionalities for supporting non-real-time collaboration. As all these functionalities are commonly supported by existing IDE products, the ATCoPE Client reuses and incorporates existing single-user programming tools and systems without reinvention (to achieve Design Objective 1).

In addition to conventional IDE functionalities and interface features, the ATCoPE Client also provides extra functionalities and features for supporting real-time and any-time collaborative programming. By means of the *ATCoPE Client Adaptor*, the real-time and

any-time collaboration functionalities are transparently integrated with the existing single-user IDE while preserving all original functionalities and user interfaces of the IDE (to achieve Design Objectives 1, 2 and 3).

2.4 Working Process and Major Functionalities

In this section, we present the design of the entire working process for collaborating programmers to interact with ATCoPE, and specify the major functionalities provided by the ATCoPE system in each stage of the working process.

2.4.1 Logging into the System: User Account and Source Code Repository Management

Similar to working with a conventional version control system, a programmer needs a user account to access the source code repository, perform version control operations, and use other collaboration services in the ATCoPE system. The NRTCOS component of the ATCoPE Server is responsible for incorporating source code repository management and version control functionalities from existing version control systems, and the ATCoPE Client Adaptor is responsible for keeping the process of using user accounts to log into the ATCoPE system the same as that of using a conventional IDE with an integrated version control client. From the perspective of ATCoPE end-users, the login process for either non-real-time or real-time collaboration is the same: using the same user account and interface to access the same source code repository managed by the same version control system in the background. After logging into the ATCoPE system, the programmer can freely access and browse the source code trees granted to his/her account.

2.4.2 Creating a Session: Start of Collaborative Work

Upon logging into the ATCoPE system, the programmer may issue a *check-out* version control command to download the selected source code files and directories to the local workspace, which triggers the creation of an ATCoPE collaboration session. The collaboration session may be either a non-real-time or a real-time session, depending on the collaboration option chosen by the programmer at the time of check-out.

If the *non-real-time collaboration mode* is chosen, the selected source code files and directories are downloaded directly from the NRTCoS version control repository to the ATCoPE Client, without any interaction with the RTCoS component. The notion of a non-real-time collaboration session is *implicit* in the sense that no explicit session information is recorded or maintained by the ATCoPE Server, and all programmers working under the non-real-time collaboration mode for the same project are implicitly belonging to the same non-real-time collaboration session.

If the *real-time collaboration mode* is chosen, the selection of the source code files and directories to be checked out, together with the user account information and authentication data, is transmitted to the RTCoS component to create a real-time collaboration session record, which contains the real-time group membership information (initially the session creator's account name and authentication data), and a cache of the source code copy being checked out. The source code copy is then downloaded and duplicated at the ATCoPE Client's workspace. The creation of a real-time collaboration session is transparent to the NRTCoS component, as the check-out process in the version control system is always the same, regardless of whether a real-time collaboration session is created or not. Each explicit real-time collaboration session managed by the RTCoS

component may correspond to an implicit non-real-time collaboration session in the NRTCOS version control system, and such correspondence facilitates loosely-coupled integration among non-real-time and real-time collaborating programmers.

For collaborating programmers, the creation of either a non-real-time or real-time collaboration session is nearly the same, except for selecting a different option at the time of checking out the source code from the version control repository. The procedures executed at the server-side are transparent to the end-user, but some distinctive visual indications can be provided in the user interface of the ATCoPE Client to differentiate whether the current collaboration session is non-real-time or real-time.

2.4.3 Joining a Session: Dynamic Session Membership Management

In addition to conventional repository interfaces for browsing and checking out source code files and directories to create new sessions, the ATCoPE system also provides a list of existing real-time collaboration sessions available for the programmer to join. Joining a non-real-time collaboration session happens *implicitly* as long as the programmer checks out the source code for the same project from the version control repository, or resumes the single-user programming work on the current working copy of the source code which has been previously checked out from the repository.

Differently, joining a real-time collaboration session may require the programmer to obtain the permission first. In the simplistic case where a real-time collaboration session is open for anyone holding a valid user account in the ATCoPE system, the programmer can immediately join the session. In a more sophisticated case where a real-time collaboration session is open for a specific group of ATCoPE end-users only, the late-comers must explicitly request and obtain the permission granted by the session creator. Once

the permission is obtained, the user will join the real-time collaboration session under the control of a *distributed join-protocol*, which has been designed for ensuring the consistency of the source code copy and supporting smooth transitions in the face of dynamic session membership changes (to be presented in Section 3.3). Upon joining a real-time collaboration session, the new session member is registered and recorded in the RTCoS component, and the latest source code copy of this real-time collaboration session (stored inside the source code cache of the RTCoS component) is downloaded and duplicated at the ATCoPE Client of the new member. At the same time, all existing members of the real-time collaboration session are notified of the joining of the new member, by means of certain indications in the user interface.

2.4.4 Working in a Session: Features of Any-Time Collaboration

In a non-real-time collaboration session, a programmer works on the source code copy inside the private workspace, and all changes made on the source code are local and invisible to the public until the local programmer commits the changes to the version control repository. To incorporate concurrent changes made by others, a programmer has to explicitly issue the *update* version control command, which merges the latest version of the shared source code copy at the version control repository with his/her local source code copy. The *update/commit* process may involve conflict detection and resolution, as conflicts may be caused by concurrent changes on shared source code files [74].

In a real-time collaboration session, a programmer also works on the source code copy inside the local workspace, but all changes made on the source code will automatically and instantly become visible to other real-time collaborating programmers without having to execute *update/commit* commands. Particularly, multiple programmers can con-

currently edit the contents of the same source code file at the same time, thus creating an internal real-time collaborative programming session within a smaller scope. To differentiate these real-time collaboration sessions within different scopes, we use the following two terms: (1) *real-time cluster session*, which is a real-time collaboration session created when programmers check out a cluster of files and directories from the version control repository; and (2) *real-time file session*, which is a real-time collaborative editing session created when multiple programmers are concurrently editing the contents of the same source code file inside a real-time cluster session.

During the sessions, programmers may freely and concurrently perform two types of editing operations: (1) *cluster-level editing operation*: to *create*, *delete* or *rename* any source code file or directory; and (2) *file-level editing operation*: to *edit* the contents of any source code file by *inserting* and *deleting* texts.

In a real-time collaboration session, the timeliness of propagating changes to other collaborating programmers depends on the natures and needs of the collaborative work, as specified below:

1. When multiple programmers are concurrently performing cluster-level editing operations to the shared source code copy in a real-time cluster session, changes made by one programmer are *instantly* propagated to all other programmers within the same real-time cluster session. If multiple concurrent operations target source code files/directories in the same or hierarchically-related directories, the operations may conflict with each other, and the conflicts will be automatically detected and resolved by underlying mechanisms and techniques (to be presented in Section 3.2.1).
2. When multiple programmers are concurrently performing file-level editing opera-

tions in a real-time file session, changes made by one programmer are *instantly* propagated to all other programmers within the same real-time file session, but only propagated *at the time of saving the file* to the programmers outside this real-time file session but within the same real-time cluster session. The delayed propagation to those programmers outside the real-time file session is reasonable as they are not working on or interested in the latest contents of the source code file at the moment. The delayed propagation also helps achieve good system performance and efficient usage of the communication bandwidth (for Design Objective 4). In a real-time file session, conflicts caused by concurrent operations on the file contents will be automatically detected and resolved by underlying techniques and mechanisms (to be discussed in Section 3.2.2).

In addition to cluster-level and file-level editing operations, a real-time collaborating programmer may also issue version control commands (e.g., *update*, *commit*) to merge the current source code copy of the real-time collaboration session with the latest copy in the version control repository. All interactions between a real-time collaboration session and the version control system are performed in the name of the user who created the real-time collaboration session. To the version control system, a real-time collaborating group is merely a single user, regardless of how many active members are actually involved in the real-time collaboration session. If conflicts with other non-real-time collaborators are detected and reported in processing the *update* or *commit* command, programmers in the same real-time collaboration session can jointly resolve the conflicts: they may separately resolve conflicts in different source code files individually, or jointly resolve conflicts in the same source code file in a real-time file session. Such loosely-coupled interaction among non-real-time and real-time collaborating program-

mers is a unique any-time collaboration feature provided by ATCoPE.

2.4.5 Leaving and Terminating a Session: Completion of Collaborative Work

Upon completion of the collaborative work, a programmer may leave the collaboration session at any time. Leaving a non-real-time collaboration session is implicit by simply committing the local source code copy into the version control repository. Leaving a real-time collaboration session can be initiated by any session member under the control of a *distributed leave-protocol*, which flushes all local changes at the leaving site to other existing sites within the session and notifies them of the leaving event (to be presented in Section 3.3). Consequently, the corresponding session membership record related to the leaving site will be updated in the RTCoS component.

After the last active member has left a real-time collaboration session, the session will be terminated and the whole session record will be removed from the RTCoS component. Before the termination of a real-time collaboration session, the latest source code copy of the session (stored inside the source code cache managed by the RTCoS component) will be committed back to the version control repository, which implicitly signifies that this real-time collaborating group has left the corresponding non-real-time collaboration session.

2.4.6 Switching among Different Collaboration Modes and Sessions: Flexible Any-Time Collaboration

In the ATCoPE system, a programmer may freely and conveniently switch among different collaboration modes and sessions at any time. A programmer may flexibly switch

(1) among different non-real-time collaboration sessions, (2) among different real-time collaboration sessions, and (3) among non-real-time and real-time collaboration sessions (thus switching among different collaboration modes). ATCoPE supports convenient switching and smooth transitions among different collaboration modes and sessions to meet the dynamic collaboration needs as follows:

1. The switching among non-real-time collaboration sessions happens implicitly when the programmer switches the active workspace from one project to another project in the programming environment under the non-real-time collaboration mode.
2. The switching among real-time collaboration sessions happens as long as a programmer leaves one real-time collaboration session and joins another real-time collaboration session. The smooth transitions among real-time collaboration sessions will be achieved by the dynamic session membership management (to be presented in Section 3.3) provided by the ATCoPE system.
3. When a programmer needs to switch from a non-real-time collaboration session to a real-time collaboration session, s/he can perform a simple operation provided in the ATCoPE Client's user interface to create a real-time collaboration session based on the current local source code copy. Internally, this will trigger the uploading of the local source code copy to the RTCoS source code cache and the creation of a new real-time collaboration session.
4. Conversely, when a programmer needs to switch from a real-time collaboration session to a non-real-time collaboration session, s/he can also perform a simple operation in the user interface to complete such transition, which makes this programmer leave the current real-time collaboration session and resume the single-user pro-

programming work based on the current source code copy in the local workspace.

2.5 Summary

In this chapter, we presented a novel *any-time collaborative programming environment* (ATCoPE) to realize the any-time collaborative programming paradigm, contributed the general design of the system architecture, and specified the functionalities for each major stage during the whole working process with ATCoPE.

At the beginning, we proposed four general design objectives for ATCoPE, which serve as the guidance for ATCoPE technical research and system implementation. The first three design objectives are concerned with the functionalities and features of ATCoPE for supporting advanced collaboration, whereas the last design objective is concerned with the system performance and scalability.

Following these general design objectives, the ATCoPE system architecture and major components were presented. The ATCoPE system consists of one ATCoPE Server and multiple ATCoPE Clients connected via communication networks. Major components inside the ATCoPE Server include the NRTCoS (Non-Real-Time Collaboration Service) component for supporting conventional non-real-time collaboration functionalities, the RTCoS (Real-Time Collaboration Service) component for supporting real-time collaboration functionalities, and the ATCoPE Server Interface for providing uniform and compatible interfaces for ATCoPE Clients to interact with the server under both non-real-time and real-time collaboration modes. The RTCoS component relies on the NRTCoS component for providing non-real-time collaboration functionalities in real-time collaboration sessions, thus facilitating the interactions among non-real-time and

real-time collaborating programmers/groups. Within the ATCoPE Client, the ATCoPE Client Adaptor has been designed for incorporating conventional single-user programming and non-real-time collaboration functionalities, providing advanced real-time collaboration functionalities, and supporting any-time collaboration features.

We specified the functionalities and features provided by the ATCoPE system at each major stage during the working process from the perspective of ATCoPE end-users. For logging into the system, the programmer uses the same user account, interface and process for both non-real-time and real-time collaborative programming. To create or join a collaboration session, the programmer follows the same procedure for both kinds of collaborative programming, and the system performs corresponding operations in the background in response to the programmer's choice on the collaboration mode. While working in a collaborative programming session, the system preserves conventional single-user and non-real-time collaborative programming functionalities, and provides advanced real-time collaboration functionalities and features. Programmers can perform non-real-time collaboration operations in a real-time collaboration session, achieving the interaction among non-real-time and real-time collaborating programmers/groups. In addition, programmers are allowed to flexibly join or leave any real-time collaboration session, and conveniently switch among collaboration modes and sessions at any time.

In the next chapter, we will present major technical issues and solutions for supporting and realizing the ATCoPE system architecture, functionalities and features designed and specified in this chapter.

Chapter 3.

Major Technical Issues and Solutions for Supporting ATCoPE

3.1 Introduction

In this chapter, we present major technical issues and solutions for supporting ATCoPE functionalities designed and specified in Chapter 2.

There are two major technical issues related to supporting real-time collaborative programming in ATCoPE. Firstly, real-time collaborative programming allows programmers to freely and concurrently perform any cluster-level or file-level editing operation at any programming target, which may lead to conflicts and inconsistencies among the shared source code copies. Secondly, the dynamic session membership management is needed to ensure smooth membership changes when programmers join or leave an ongoing real-time collaboration session. These two issues will be addressed in Section 3.2 and Section 3.3 respectively.

In supporting non-real-time collaborative programming in ATCoPE, there are also two major technical issues. Firstly, it is needed to provide programmers with conventional non-real-time collaboration functionalities under the non-real-time collaboration mode. Secondly, it is also needed to support the use of non-real-time collaboration functionalities in real-time collaboration sessions so as to facilitate the loosely-coupled interactions among non-real-time collaborating programmers and real-time collaborating groups, which is a unique feature of any-time collaborative programming. The common issue in meeting these two requirements is to incorporate conventional non-real-time collabora-

tion functionalities into the ATCoPE system, which will be presented in Section 3.4.

3.2 Supporting Consistency Maintenance in Real-Time Collaboration Sessions

To achieve high local responsiveness and fast remote notification/merging (as specified in Design Objective 4) in real-time collaboration sessions over high latency communication networks such as the Internet, the ATCoPE system has been designed with a *replicated architecture*: the shared source code copy (i.e., the cluster of source code files and directories) and the ATCoPE Client application are replicated at each real-time collaborating site. Any editing operation performed on the shared source code file or directory is immediately executed on the local replica (thus achieving high local responsiveness), and then instantly propagated to all remote sites (within the same session) for execution (thus achieving fast remote notification and merging). Such architecture also enables *unconstrained* real-time collaboration in the sense that collaborating programmers can freely and concurrently access and update any part of the source code copy at any time without any constraint.

One technical challenge in supporting unconstrained real-time collaborative programming over the replicated architecture is the consistency maintenance, i.e., after all editing operations are propagated and executed at all real-time collaborating sites, the distributed replicas of the source code copy must be identical across all collaborating sites. Since there are two types of real-time collaboration sessions and editing operations (i.e., cluster-level editing operations in real-time cluster sessions and file-level editing operations in real-time file sessions), two sets of consistency maintenance techniques have been devised, one for each of them.

3.2.1 Consistency Maintenance in Real-Time Cluster Sessions

Concurrent cluster-level editing operations may conflict with each other if they target files/directories in the same directory or hierarchically-related directories. For example, if two programmers concurrently rename the same file to different new names, both local operations will be executed successfully at their local sites, but problems will occur when the two editing operations arrive at remote sites for execution: when the ATCoPE system attempts to replay the remote *Rename* operation, the targeted file does not exist (because it has already been renamed by the local site), leading to the failure of the execution and resulting in inconsistent source code copies.

One approach to solving this problem is to use turning-taking or locking-based protocols to prevent concurrent editing operations [34][35][46]. Under such approaches, a collaborating site must obtain the *permission token* from a centralized coordinator before performing any cluster-level editing operation, and the token must be returned to the coordinator after the editing operation has been performed locally and executed at all remote sites. This approach achieves consistency by prohibiting concurrent cluster-level editing operations, thus eliminating the possibility of operation conflicts. However, such *pessimistic approach* is too restrictive in supporting real-time collaboration which requires unconstrained interaction, because it may slow-down the local responsiveness as a local operation cannot be executed before the permission token is obtained, and it may incur high communication overheads as each cluster-level editing operation will need a global synchronization procedure, which is costly in high-latency communication networks.

To support unconstrained collaboration and achieve high performance and scalability

for the ATCoPE system (as specified in Design Objective 4), an *optimistic approach* to consistency maintenance has been adopted for cluster-level editing operations. Under the optimistic approach, all collaborating programmers in a real-time cluster session are allowed to perform any cluster-level editing operation at any target without having to request and obtain permission, and the system automatically detects and resolves operation conflicts when executing operations propagated from remote sites. Such kind of conflict resolution and consistency maintenance approach has been used for supporting various kinds of collaborative editing systems in prior work [100], such as the dependency conflict resolution in real-time collaborative 3D design systems [3][6][8] and the orthogonal conflict resolution in real-time collaborative 2D editing systems [97].

Before presenting the supporting techniques for conflict detection and resolution mechanisms, we first provide the general definitions for conflict and compatible relationships among cluster-level editing operations, present the conflict and compatible relationships among all pairs of cluster-level editing operations, and specify the conflict resolution results for all pairs of conflicting operations.

Definition 3.1: Conflict Relationship between Cluster-Level Editing Operations

Given two cluster-level editing operations O_1 and O_2 performed on the same source code cluster, they have a conflict relationship, denoted as $O_1 \otimes O_2$, if and only if: (i) O_1 and O_2 are concurrent; and (ii) different execution orders of O_1 and O_2 result in inconsistent source code clusters.

Definition 3.2: Compatible Relationship between Cluster-Level Editing Operations

Given two cluster-level editing operations O_1 and O_2 performed on the same source code cluster, they have a compatible relationship, denoted as $O_1 \odot O_2$, if and only

if they do not have a conflict relationship, i.e., $\neg(O_1 \otimes O_2)$.

There are four primitive cluster-level editing operations that can be executed on a cluster of source code files and directories in a real-time collaboration session:

1. *Create*(N, D): to create a source code file/directory named N in the directory D ;
2. *Delete*(N, D): to delete the source code file/directory named N in the directory D ;
3. *Rename*(ON, NN, D): to rename the source code file/directory named ON in the directory D to the new name NN ; and
4. *Edit*(N, D): to edit the contents of the source code file named N in the directory D .

Based on comprehensive analysis, the conflict/compatible relationships among all pairs of cluster-level editing operations have been derived as follows.

Firstly, two concurrent cluster-level editing operations that target files/directories in the same directory are conflicting under the following conditions:

1. A *Create*(N_1, D) conflicts with another *Create*(N_2, D) if and only if both operations create a new file/directory with the same name, i.e., $N_1 = N_2$.
2. A *Create*(N_1, D) conflicts with a *Rename*(ON_2, NN_2, D) if and only if the created file/directory has the same name with the renamed file/directory, i.e., $N_1 = NN_2$.
3. A *Delete*(N_1, D) conflicts with a *Rename*(ON_2, NN_2, D) if and only if the *Delete* operation deletes the file/directory that is to be renamed by the *Rename* operation, i.e., $N_1 = ON_2$.
4. A *Delete*(N_1, D) conflicts with an *Edit*(N_2, D) if and only if the *Delete* operation

deletes a source code file in which the contents are to be edited by the *Edit* operation, i.e., $N_1 = N_2$.

5. A $Rename(ON_1, NN_1, D)$ conflicts with another $Rename(ON_2, NN_2, D)$ if and only if:
 - (i) The two operations rename the same file/directory to different new names, i.e., $(ON_1 = ON_2) \wedge (NN_1 \neq NN_2)$; or
 - (ii) The two operations rename two different files/directories to the same new name, i.e., $(ON_1 \neq ON_2) \wedge (NN_1 = NN_2)$.
6. A $Rename(ON_1, NN_1, D)$ conflicts with an $Edit(N_2, D)$ if and only if the *Rename* operation renames a source code file in which the contents are to be edited by the *Edit* operation, i.e., $ON_1 = N_2$.

Secondly, two concurrent cluster-level editing operations that target hierarchically-related file/directories are conflicting under the following conditions, where $D_1 \neq D_2$:

1. A $Delete(N_1, D_1)$ conflicts with a $Create(N_2, D_2)$ if and only if the *Create* operation creates a file/directory underneath⁴⁰ the directory that is to be deleted by the *Delete* operation, i.e., $D_2 \subseteq N_1$.
2. A $Rename(ON_1, NN_1, D_1)$ conflicts with a $Create(N_2, D_2)$ if and only if the *Create*

⁴⁰ One file/directory in a directory D_1 is underneath another directory D_2 , denoted as $D_1 \subseteq D_2$, if and only if (i) $D_1 = D_2$ or (ii) D_1 is a sub-directory of D_2 (via one or multiple levels).

operation creates a file/directory underneath the directory that is to be renamed by the *Rename* operation, i.e., $D_2 \subseteq ON_1$.

3. A *Delete*(N_1, D_1) conflicts with a *Rename*(ON_2, NN_2, D_2) if and only if:
 - (i) The *Rename* operation renames a file/directory underneath the directory that is to be deleted by the *Delete* operation, i.e., $D_2 \subseteq N_1$; or
 - (ii) The *Delete* operation deletes a file/directory underneath the directory that is to be renamed by the *Rename* operation, i.e., $D_1 \subseteq ON_2$.
4. A *Delete*(N_1, D_1) conflicts with an *Edit*(N_2, D_2) if and only if the *Edit* operation is to edit the contents in a source code file underneath the directory that is to be deleted by the *Delete* operation, i.e., $D_2 \subseteq N_1$.
5. A *Rename*(ON_1, NN_1, D_1) conflicts with another *Rename*(ON_2, NN_2, D_2) if and only if one operation renames a file/directory underneath the directory that is to be renamed by the other operation, i.e., $(D_1 \subseteq ON_2) \vee (D_2 \subseteq ON_1)$.
6. A *Rename*(ON_1, NN_1, D_1) conflicts with an *Edit*(N_2, D_2) if and only if the *Edit* operation is to edit the contents in a source code file underneath the directory that is to be renamed by the *Rename* operation, i.e., $D_2 \subseteq ON_1$.

The above pairs of operations are compatible under other non-specified conditions, and other pairs of operations are always compatible under any condition. There are three pairs of compatible operations that are worth discussing:

1. Two concurrent operations *Delete*(N_1, D) and *Delete*(N_2, D) are always compatible even though they target the same file/directory (i.e. $N_1 = N_2$) in the same directory.

This is because their operation intentions/effects are exactly the same. Technically, the later executed *Delete* operation will fail because the targeted file/directory has already been deleted by the prior executed *Delete* operation.

2. Similar to the first point, two concurrent operations $Delete(N_1, D_1)$ and $Delete(N_2, D_2)$ are always compatible even though $Delete(N_1, D_1)$ deletes a file/directory underneath the directory that is to be deleted by $Delete(N_2, D_2)$, i.e., $D_1 \subseteq N_2$. This is because their operation intentions do not conflict. Technically, if $Delete(N_1, D_1)$ is executed after $Delete(N_2, D_2)$, the execution of $Delete(N_1, D_1)$ will fail because the targeted N_1 , which is in D_1 underneath N_2 , has already been deleted by the prior execution of $Delete(N_2, D_2)$.
3. Two concurrent operations $Edit(N_1, D)$ and $Edit(N_2, D)$ are always compatible even though they are to start editing the contents of the same source code file, i.e., $N_1 = N_2$. Conceptually, concurrent editing operations on the contents of the same source code file are not conflicting in unconstrained real-time collaborative editing environments. Technically, in case that two collaborating sites concurrently start to edit the contents of the same source code file (i.e., two collaborating sites concurrently joining the same real-time file session), the case will be handled by the specific distributed computing technique (to be presented in Section 3.3), which ensures the consistency of the source code copy while accommodating concurrent joining.

The conflict/compatible relationships and the conflict conditions between all pairs of cluster-level editing operations are summarized in Table 3.1 and Table 3.2 as follows.

Table 3.1: Conflict/compatible relationships between cluster-level editing operations targeting files/directories in the same directory

	$Create(N_2, D)$	$Delete(N_2, D)$	$Rename(ON_2, NN_2, D)$	$Edit(N_2, D)$
$Create(N_1, D)$	$\otimes \Leftrightarrow N_1 = N_2$	$\odot$	$\otimes \Leftrightarrow N_1 = NN_2$	$\odot$
$Delete(N_1, D)$		$\odot$	$\otimes \Leftrightarrow N_1 = ON_2$	$\otimes \Leftrightarrow N_1 = N_2$
$Rename(ON_1, NN_1, D)$			$\otimes \Leftrightarrow [(ON_1 = ON_2) \wedge (NN_1 \neq NN_2)] \vee [(ON_1 \neq ON_2) \wedge (NN_1 = NN_2)]$	$\otimes \Leftrightarrow ON_1 = N_2$
$Edit(N_1, D)$				$\odot$

Table 3.2: Conflict/compatible relationships between cluster-level editing operations targeting hierarchically-related file/directories ($D_1 \neq D_2$)

	$Create(N_2, D_2)$	$Delete(N_2, D_2)$	$Rename(ON_2, NN_2, D_2)$	$Edit(N_2, D_2)$
$Create(N_1, D_1)$	$\odot$	$\otimes \Leftrightarrow D_1 \subseteq N_2$	$\otimes \Leftrightarrow D_1 \subseteq ON_2$	$\odot$
$Delete(N_1, D_1)$		$\odot$	$\otimes \Leftrightarrow (D_2 \subseteq N_1) \vee (D_1 \subseteq ON_2)$	$\otimes \Leftrightarrow D_2 \subseteq N_1$
$Rename(ON_1, NN_1, D_1)$			$\otimes \Leftrightarrow (D_1 \subseteq ON_2) \vee (D_2 \subseteq ON_1)$	$\otimes \Leftrightarrow D_2 \subseteq ON_1$
$Edit(N_1, D_1)$				$\odot$

Given the derived conflict relationships, we specify the conflict resolution results for all pairs of conflicting operations in Table 3.3 and Table 3.4. One major design rationale for the conflict resolution results is to preserve the effects of both conflicting operations (as combined effects) and to avoid the loss of work as much as possible.

Table 3.3: Specifications of resolution results for conflicting operations
targeting files/directories in the same directory

Conflicting Operations	Conflict Resolution Results
$Create(N_1, D) \otimes Create(N_2, D)$ (Issued by P_1 and P_2 respectively)	<p>Two files/directories are created with proper and unique names: the two items will be <i>automatically renamed</i> to $[N_1_P_1]$ and $[N_2_P_2]$ respectively.</p> <p>The resolution result preserves both operations' effects, and makes involved programmers aware of the conflict issue by the <i>automatic renaming mechanism</i>.</p>
$Create(N_1, D) \otimes Rename(ON_2, NN_2, D)$ (Issued by P_1 and P_2 respectively)	<p>A new item is created and the renamed item is kept, and both items are automatically renamed with proper and unique names: the created item will be renamed to $[N_1_P_1]$, while the renamed item will be further renamed to $[NN_2_P_2]$.</p> <p>The design rationale is the same as the conflict resolution result for $Create(N_1, D) \otimes Create(N_2, D)$.</p>
$Delete(N_1, D) \otimes Rename(ON_2, NN_2, D)$ (Issued by P_1 and P_2 respectively)	<p>The targeted file/directory $N_1 = ON_2$ is renamed and kept:</p> <ul style="list-style-type: none"> • If $Rename(ON_2, NN_2, D)$ is executed after $Delete(N_1, D)$, N_1 is restored and renamed to NN_2. • If $Delete(N_1, D)$ is executed after $Rename(ON_2, NN_2, D)$, N_1 is not deleted. <p>The resolution result conceptually preserves both operations' effects: the file/directory with the original</p>

	name is deleted and renamed at the same time.
$Delete(N_1, D) \otimes Edit(N_2, D)$ (Issued by P_1 and P_2 respectively)	<p>The file $N_1 = N_2$ is kept (not deleted) and updated with the editing operations:</p> <ul style="list-style-type: none"> • If $Edit(N_2, D)$ is executed after $Delete(N_1, D)$, N_1 is restored and being updated with the remote editing of P_2. • If $Delete(N_1, D)$ is executed after $Edit(N_2, D)$, N_2 is prevented from being deleted (while being edited). <p>The resolution result prevents the loss of work.</p>
$Rename(ON_1, NN_1, D) \otimes Rename(ON_2, NN_2, D)$ (Issued by P_1 and P_2 respectively)	<p>For conflict condition $(ON_1 = ON_2) \wedge (NN_1 \neq NN_2)$: two copies of the file/directory with different names are preserved (duplicated):</p> <ul style="list-style-type: none"> • If $Rename(ON_2, NN_2, D)$ is executed after $Rename(ON_1, NN_1, D)$, NN_1 is kept and NN_2 is created. • If $Rename(ON_1, NN_1, D)$ is executed after $Rename(ON_2, NN_2, D)$, NN_2 is kept and NN_1 is created. <p>For conflict condition $(ON_1 \neq ON_2) \wedge (NN_1 = NN_2)$: both files/directories are preserved and further renamed with proper and unique names by the abovementioned <i>automatic renaming mechanism</i>:</p> <ul style="list-style-type: none"> • If $Rename(ON_2, NN_2, D)$ is executed after $Rename(ON_1, NN_1, D)$, NN_1 is kept and renamed to $[NN_1_P_1]$, and ON_2 is renamed to $[NN_2_P_2]$.

	<ul style="list-style-type: none"> • If $Rename(ON_1, NN_1, D)$ is executed after $Rename(ON_2, NN_2, D)$, NN_2 is kept and renamed to $[NN_2_P_2]$, and ON_1 is renamed to $[NN_1_P_1]$. <p>These resolution results preserve both <i>Rename</i> operations' effects, and make involved programmers aware of the conflict issues.</p>
$Rename(ON_1, NN_1, D) \otimes Edit(N_2, D)$ (Issued by P_1 and P_2 respectively)	<p>The renamed file is kept/created and the file with the original name is being edited (without being renamed):</p> <ul style="list-style-type: none"> • If $Edit(N_2, D)$ is executed after $Rename(ON_1, NN_1, D)$, ON_1 is renamed to NN_1, and N_2 is additionally created and being updated with the remote editing of P_2. • If $Rename(ON_1, NN_1, D)$ is executed after $Edit(N_2, D)$, N_2 is prevented from being renamed, and NN_1 is additionally created. <p>The resolution result preserves both operations' effects: the file with the original name is renamed and being edited at the same time.</p>

Table 3.4: Specifications of resolution results for conflicting operations targeting hierarchically-related file/directories ($D_1 \neq D_2$)

Conflicting Operations	Conflict Resolution Results
$Delete(N_1, D_1) \otimes Create(N_2, D_2)$ (Issued by P_1 and P_2 respectively)	<p>All original files/directories underneath N_1 are deleted, but N_1 itself is retained and N_2 is created in D_2:</p> <ul style="list-style-type: none"> • If $Create(N_2, D_2)$ is executed after $Delete(N_1, D_1)$,

respectively)	<p>N_1 is restored (but without restoring original files and sub-directories underneath N_1), and N_2 is created underneath N_1.</p> <ul style="list-style-type: none"> • If $Delete(N_1, D_1)$ is executed after $Create(N_2, D_2)$, both N_2 (newly created) and N_1 are preserved, but all other files and sub-directories underneath N_1 are deleted. <p>The resolution result preserves both <i>Delete</i> and <i>Create</i> operations' effects.</p>
<p>$Rename(ON_1, NN_1, D_1)$ $\otimes Create(N_2, D_2)$</p> <p>(Issued by P_1 and P_2 respectively)</p>	<p>ON_1 is renamed to NN_1 and N_2 is created underneath NN_1 (as renamed):</p> <ul style="list-style-type: none"> • If $Create(N_2, D_2)$ is executed after $Rename(ON_1, NN_1, D_1)$, N_2 is created underneath NN_1 (as renamed). • If $Rename(ON_1, NN_1, D_1)$ is executed after $Create(N_2, D_2)$, ON_1 is renamed to NN_1. <p>The resolution result preserves both <i>Rename</i> and <i>Create</i> operations' effects.</p>
<p>$Delete(N_1, D_1) \otimes$ $Rename(ON_2, NN_2, D_2)$</p> <p>(Issued by P_1 and P_2 respectively)</p>	<p>For conflict condition $D_2 \subseteq N_1$: N_1 is restored without its original non-renamed files and sub-directories, and ON_2 is preserved and renamed to NN_2 underneath N_1:</p> <ul style="list-style-type: none"> • If $Rename(ON_2, NN_2, D_2)$ is executed after $Delete(N_1, D_1)$, N_1 is restored (but without restoring original files and sub-directories underneath N_1), and NN_2 is created underneath N_1. • If $Delete(N_1, D_1)$ is executed after $Rename(ON_2, NN_2, D_2)$, N_1 is restored (but without restoring original files and sub-directories underneath N_1), and NN_2 is created underneath N_1.

	<p>NN_2, D_2), both N_1 and NN_2 are preserved, but other non-renamed files and sub-directories underneath N_1 are deleted.</p> <p>For conflict condition $D_1 \subseteq ON_2$: N_1 is deleted underneath NN_2 (as renamed):</p> <ul style="list-style-type: none"> • If $Rename(ON_2, NN_2, D_2)$ is executed after $Delete(N_1, D_1)$, N_1 is deleted, and ON_2 is renamed to NN_2. • If $Delete(N_1, D_1)$ is executed after $Rename(ON_2, NN_2, D_2)$, ON_2 is renamed to NN_2, and N_1 is deleted underneath NN_2 (as renamed). <p>These resolution results preserve both <i>Delete</i> and <i>Rename</i> operations' effects.</p>
<p>$Delete(N_1, D_1) \otimes Edit(N_2, D_2)$</p> <p>(Issued by P_1 and P_2 respectively)</p>	<p>The source code file is kept (not deleted) and being updated with the editing operations:</p> <ul style="list-style-type: none"> • If $Edit(N_2, D_2)$ is executed after $Delete(N_1, D_1)$, N_1 is restored (but without restoring original files and sub-directories underneath N_1), and N_2 is created underneath the restored N_1 and being updated with the remote editing of P_2. • If $Delete(N_1, D_1)$ is executed after $Edit(N_2, D_2)$, both N_1 and N_2 (being edited) are preserved, but all other files and sub-directories underneath N_1 are deleted. <p>The resolution result preserves both operations' effects and prevents the loss of work.</p>

$Rename(ON_1, NN_1, D_1)$ $\otimes Rename(ON_2, NN_2, D_2)$ (Issued by P_1 and P_2 respectively)	<p>ON_2 is renamed to NN_2 underneath NN_1 (as renamed):</p> <ul style="list-style-type: none"> • If $Rename(ON_2, NN_2, D_2)$ is executed after $Rename(ON_1, NN_1, D_1)$, ON_2 is renamed to NN_2 underneath NN_1 (as renamed). • If $Rename(ON_1, NN_1, D_1)$ is executed after $Rename(ON_2, NN_2, D_2)$, ON_2 is renamed to NN_2, and ON_1 is renamed to NN_1. <p>The resolution result preserves both <i>Rename</i> operations' effects.</p>
$Rename(ON_1, NN_1, D_1)$ $\otimes Edit(N_2, D_2)$ (Issued by P_1 and P_2 respectively)	<p>The targeted directory is renamed and the source code file is being edited:</p> <ul style="list-style-type: none"> • If $Edit(N_2, D_2)$ is executed after $Rename(ON_1, NN_1, D_1)$, N_2 is being updated with the remote editing of P_2 underneath NN_1 (as renamed). • If $Rename(ON_1, NN_1, D_1)$ is executed after $Edit(N_2, D_2)$, ON_1 is renamed to NN_1 while N_2 is being edited. <p>The resolution result preserves both operations' effects.</p>

To detect conflicts among cluster-level editing operations and resolve them according to these specifications, the following techniques have been devised and applied.

Firstly, a scheme based on state vectors [95] has been devised to timestamp each cluster-level editing operation with a state vector, which is used to derive whether two editing operations are concurrent or not, as concurrency is one necessary condition for operation conflicts.

Secondly, an operation history buffer is maintained at each collaborating site to save all cluster-level editing operations that have been executed at the site. Whenever a remote cluster-level editing operation arrives and becomes causally ready for execution, operations saved in the history buffer will be scanned to check whether they are concurrent with the newly arrived operation by comparing their timestamps. If an executed operation is found to be concurrent with the newly arrived one, the detailed operation information (such as the operation type and the pathname of the targeted file/directory) is further examined to check whether the concurrent operations are conflicting with each other according to the conflict conditions specified in Table 3.1 and Table 3.2. In case that a conflict is detected, the corresponding conflict resolution procedure will be executed according to the specifications in Table 3.3 and Table 3.4.

3.2.2 Consistency Maintenance in Real-Time File Sessions

In a real-time file session, consistency maintenance is concerned with the textual contents of the shared source code file that is being collaboratively edited by multiple programmers. The textual contents of a source code file are organized as a string of characters with a linear address model, and the position of each character ranges from 0 to $(L - 1)$, where L is the length (i.e., number of characters) of the source code file. There are two primitive file-level editing operations that can be executed on the source code file:

1. *Insert*[pos , str]: to *insert* a string str at the position pos of the source code contents.
2. *Delete*[pos , len]: to *delete* a string from the starting position pos with the length of len .

Concurrent file-level editing operations may conflict with each other and lead to incon-

sistent source code contents. To illustrate, consider the example in Figure 3.1. There are two collaborating programmers (P_1 and P_2) concurrently editing a shared source code file, which initially contains a method named *TestProgram(int x)*. In this example, P_1 issues the editing operation *Insert*[0, “My”] to insert a string “My” at the beginning of the method name, while P_2 concurrently issues the editing operation *Delete*[4, 7] to delete the string “Program” from the method name. After the execution of the two concurrent file-level editing operations in different orders, the final source code contents are inconsistent. The root of the inconsistency is that the prior execution of the concurrent editing operation *Insert*[0, “My”] at P_1 ’s site has shifted the string “Program” to the right by 2 positions, and the follow-up execution of *Delete*[4, 7] on the evolved document state, which is different from the initial state on which *Delete*[4, 7] was defined, will delete the wrong characters.

Figure 3.1: An example of the conflict among file-level editing operations

The above inconsistency due to concurrent file-level editing operations is a well-known problem in real-time collaborative editing and can be resolved by using the operational

transformation (OT) technique [94][95][99]. The OT technique has been invented to maintain consistency without restricting user interactions, which is able to achieve two fundamental consistency requirements [95]: (1) *convergence*: all replicas of a shared document (under the replicated architecture) must be identical after executing the same group of operations at all sites within a real-time collaborative editing session; and (2) *intention preservation*: the effect of an operation at all replicas must be the same as its effect at the local replica. With the support of other distributed computing techniques (e.g., time-stamping editing operations by state vectors [95]), the real-time collaborative editing system also ensures *causality preservation* to ensure that editing operations are always executed in their *cause-effect* orders at all collaborating sites [56][77]. With the support of the OT technique, the conflict in Figure 3.1 can be resolved as presented in Figure 3.2.

Figure 3.2: OT-based conflict resolution for file-level editing operations

The remote editing operation `Delete[4, 7]`, which arrives at P_1 's site, will be transformed against the concurrent operation `Insert[0, "My"]` to produce a transformed operation

Delete[6, 7], which increases the position parameter from 4 to 6 for compensating the positional shifting effect caused by *Insert*[0, “My”]. Execution of the transformed operation *Delete*[6, 7] will correctly delete the intended string “*Program*”, and achieve correct and consistent results. In essence, the OT technique transforms parameters of concurrent file-level editing operations to compensate the positional shifting effects (in the domain of plain text editing), so that their executions in different orders can produce consistent and intended results.

3.3 Supporting Dynamic Session Membership in Real-Time Collaboration Sessions

In any-time collaborative programming, programmers are able to switch among different collaboration sessions and modes spontaneously to meet dynamic and varying collaboration needs. Joining or leaving a non-real-time collaboration session happens implicitly with the interaction between the version control system and a programmer (or a real-time collaborating group in the name of one user), and thus the dynamic session membership for non-real-time collaboration sessions is naturally supported by the version control system. Differently, additional techniques are needed for achieving dynamic session membership for real-time collaboration sessions, which allows programmers to freely join and leave any real-time collaboration session without any constraint.

For supporting dynamic session membership, a real-time collaboration session should be able to accept a new session member at any time while other existing members are currently working in the ongoing session. The ATCoPE system needs to pass the latest source code copy of the session to the new member and ensure that upon the joining of the new member, all members will resume the real-time collaborative programming

work based on a consistent copy of the source code. In addition, this transition procedure should be completed with minimum overheads and inconvenience brought to collaborating programmers. To accommodate such dynamic session membership changes in a reliable way, a *distributed join-protocol* has been designed. Major steps of the join-protocol procedure are listed and explained below, and an example of the procedure is presented in Figure 3.3 to illustrate how the join-protocol works step by step in a real-time collaboration session with two existing members (Member 1 and Member 2) for accepting a new member (Member 3) to join the session.

Figure 3.3: The distributed join-protocol for dynamic session membership

1. [Client-Side] When a new member attempts to join a real-time collaboration session (assuming that the member holds the permission to work in the session), the ATCoPE Client of this new member sends a *JOIN* message to the session manager of the ATCoPE Server to request joining an existing real-time collaboration session. After sending the *JOIN* message, the new member is not allowed to perform any local editing operation until the join-protocol procedure is completed.
2. [Server-Side] Upon receiving the *JOIN* message, the session manager broadcasts a *START* message to all existing ATCoPE Clients in the real-time collaboration session to inform them of the start of the join-protocol procedure.

3. [Client-Side] Upon receiving the *START* message sent from the session manager, an ATCoPE Client completes all ongoing operations, prevents the generation of new local operations (but still allowing the execution of remote operations for ensuring the consistency of the source code copy), and sends back a *READY* message to the session manager to inform its readiness for entering the *quiescent* state.
4. [Server-Side] Upon receiving *READY* messages from all existing members' clients, the session manager broadcasts a *FINISH* message to all (existing and new) members' clients to inform the completion of the join-protocol procedure. In the meantime, the session manager registers the new member in the session's *active membership list* managed by the RTCoS component of the ATCoPE Server.
5. [Client-Side] Upon receiving the *FINISH* message sent from the session manager, an ATCoPE Client continues to allow normal editing work. In addition to the *FINISH* message, the new member's ATCoPE Client will also receive the latest source code copy of the session, which is stored in the source code cache managed by the RTCoS component of the ATCoPE Server, and consistent with source code copies located at all existing members' ATCoPE Clients.

In the ATCoPE system, all communications between ATCoPE Clients and the ATCoPE Server are performed in FIFO (First In First Out) channels (e.g., TCP connections), and therefore the real-time collaboration session will enter the *quiescent* state when all clients have received the *FINISH* message. In a quiescent state, there is no message in transition, and all clients must have executed the same set of editing operations and hence have consistent copies of the source code. At the end of the join-protocol procedure, the new member will receive the latest source code copy, and all members will

resume the programming work based on this consistent source code copy, as if a fresh session is started.

Multiple ATCoPE Clients are allowed to concurrently send joining requests for the same real-time collaboration session, but the real-time session manager serializes those requests to avoid inconsistency and complication: if there is an ongoing join-protocol procedure for the same session, a new request will be queued until the ongoing procedure is completed. However, concurrent joining requests for different sessions can be processed in parallel, and the ATCoPE Server has the capability of handling multiple join-protocol procedures simultaneously.

In a real-time collaboration session, existing members can also freely leave the session at any time. As presented in Figure 3.4, a distributed leave-protocol has been designed. When an existing member attempts to leave the session, the ATCoPE Client sends a *LEAVE* message to the session manager. Upon receiving a leaving request from an existing client, the session manager removes the leaving client from the session's active membership list, closes the network connection with the client, and broadcasts a *LEAVE* message to other remaining members' clients to inform them of the leaving event.

Figure 3.4: Distributed leave-protocol for dynamic session membership

3.4 Supporting Non-Real-Time Collaboration Functionalities in ATCoPE

As specified by the ATCoPE system architecture in Section 2.3, the NRTCoS component provides non-real-time collaboration functionalities in terms of user account management, source code repository management and source code versioning management. To preserve the compatibility with conventional non-real-time collaborative programming tools and environments, the NRTCoS component supports these functionalities by incorporating existing version control systems.

With the NRTCoS component as the core, multiple components within the ATCoPE system collectively support non-real-time collaboration functionalities in two aspects: (1) incorporating non-real-time collaboration services for supporting non-real-time collaborative programming; and (2) bridging non-real-time collaboration services and real-time collaboration sessions for facilitating interactions among non-real-time and real-time collaborating programmers/groups. The following two sub-sections will present how various components of the ATCoPE system cooperate with each other in supporting these functionalities.

3.4.1 Incorporating Non-Real-Time Collaboration Services for Non-Real-Time Collaborative Programming

For conducting non-real-time collaborative programming, programmers are able to use the ATCoPE Client to perform version control operations (such as issuing *check-out*, *update* and *commit* commands) in the same way as using a conventional IDE with a version control client. As presented in Figure 3.5, the ATCoPE Server Interface passes these version control operations to the NRTCoS component, which in turn invokes pub-

lic interfaces of the incorporated version control system to execute these operations.

Figure 3.5: Incorporating non-real-time collaboration services for non-real-time collaborative programming

At the beginning of the non-real-time collaborative programming work, the source code copy is transmitted from the NRTCOS source code repository to the local workspace of the ATCoPE Client by executing the *check-out* command. By the *update* command, the source code copy is downloaded from the NRTCOS source code repository to the ATCoPE Client's local workspace via the ATCoPE Server Interface; by the *commit* command, the source code copy is uploaded from the local workspace of the ATCoPE Client to the NRTCOS source code repository via the ATCoPE Server Interface.

3.4.2 Bridging Non-Real-Time Collaboration Services and Real-Time Collaboration Sessions

The RTCOS component of the ATCoPE Server maintains a corresponding source code cache for each real-time collaboration session, which stores the latest source code copy as a shared repository within the session. The source code cache is located at a particu-

lar physical location of the server's data storage, while the RTCoS component maintains the correct mapping between real-time collaboration sessions and source code caches. The source code cache is designed and maintained for several purposes. Firstly, it serves as a reliable storage for the working source code copy of the session, which can be used for data recovery in case that a session encounters communication breakdown or server application crash issues. Secondly, it is able to support the fast resuming of a real-time collaboration session after an inactive period without retrieving the source code copy from the version control repository, which is similar to the first purpose. Thirdly, whenever a new client joins a session, the latest source code copy can be directly transmitted from the source code cache to the new client without involving other clients for passing the source code copy, thus reducing the workload and delay at existing clients (i.e., existing clients can immediately resume the normal editing work after receiving the *FINISH* message). Fourthly, in processing a *commit* version control command within a real-time collaboration session, the latest source code copy can also be directly uploaded from the source code cache to the version control repository without involving clients for passing the data, which is similar to the third purpose.

As presented in Figure 3.6, when an ATCoPE end-user creates a real-time collaboration session, the source code copy is checked out from the NRTCoS source code repository to the corresponding source code cache for the newly created real-time collaboration session in the RTCoS component. The source code copy is then downloaded from the RTCoS source code cache to the ATCoPE Client's local workspace via the ATCoPE Server Interface. Later on, when a new member joins the real-time collaboration session, the latest source code copy of the session is transmitted from the RTCoS source code cache to the new member's local workspace upon the completion of the join-protocol

procedure presented in Section 3.3.

Figure 3.6: Bridging non-real-time collaboration services and real-time collaboration sessions

During a real-time collaboration session, programmers concurrently perform editing operations on the source code copy, and all changes are instantly propagated to other sites for real-time remote execution and applied in the RTCoS source code cache as well. A propagated cluster-level editing operation is immediately executed in the source code cache, whereas file-level editing operations propagated via the server are not executed in the cache. The contents of a source code file in the cache will only be updated when

any site issues a *Save* command to the source code file in the real-time file session (in correspondence with the design in Section 2.4.4). The source code copy maintained in the cache is consistent with all source code copies distributed at the local workspaces of ATCoPE Clients after all editing operations have been propagated/executed at all sites and all source code files being edited have been saved within the session.

For supporting real-time collaborating programmers to use non-real-time collaboration services, each real-time collaboration session is associated with a *Non-Real-Time Collaboration Proxy* maintained by the RTCoS component, which bridges non-real-time collaboration services and real-time collaboration sessions. Whenever a session member issues a version control operation such as an *update/commit* command on behalf of the group, this operation is transmitted to the corresponding Non-Real-Time Collaboration Proxy inside the RTCoS component, which further relays these operations (and associated data) to the NRTCoS component.

As illustrated in Figure 3.6, in processing an *update* command, the latest source code copy in the NRTCoS source code repository is downloaded and merged with the current source code copy stored in the RTCoS source code cache, and the updates are further downloaded to the local workspaces of all ATCoPE Clients within the real-time collaboration session. Conversely, in processing a *commit* command, the latest source code copy stored in the RTCoS source code cache is committed to the NRTCoS source code repository via the Non-Real-Time Collaboration Proxy of the session.

The Non-Real-Time Collaboration Proxy represents the real-time collaborating group and acts as a single client (in the name of the session creator) of the NRTCoS component. Therefore, the credential data of the session creator is also saved in the RTCoS

real-time session management module at the time of the session creation for facilitating authentication when communicating with the NRTCOS component.

In addition, while processing non-real-time collaboration functions within a real-time collaboration session, the system must also ensure the consistency of the shared source code copy in the session. This has been achieved by using synchronization protocols to force the session to enter a quiescent and consistent state before processing version control operations. For supporting the process of *update* and *commit* commands in real-time collaboration sessions, two distributed synchronization protocols have been devised respectively, which are similar to the distributed join-protocol in Section 3.3 by replacing the *JOIN* message with an *UPDATE* or *COMMIT* message. When the session reaches a quiescence state, the shared source code copy is consistent across all sites as well as the RTCOS source code cache, and this latest source code copy is either uploaded to the NRTCOS source code repository by the *commit* command or merged with the downloaded copy from the NRTCOS source code repository by the *update* command.

During the execution of either an *update* or *commit* command, conflicts may be detected and reported by the version control system in the NRTCOS component, and the conflict resolution can be conducted in a real-time collaboration fashion within the session. Moreover, in the face of concurrent version control operations issued by multiple active members in the same real-time collaboration session, they are serialized by the real-time session manager to avoid inconsistency and complication.

3.5 Summary

In this chapter, we presented three major sets of technical issues and solutions for sup-

porting key functionalities of ATCoPE specified in Chapter 2. All solutions have been devised in compliance with the general design objectives proposed in Section 2.2.

Firstly, for supporting real-time collaboration with high local responsiveness and fast remote notification/merging, the replicated architecture has been adopted, and the system allows all local editing operations to be executed at the local site without any delay or constraint, and automatically detects and resolves operation conflicts in case of concurrency when the operation arrives at a remote site. Consequently, the consistency maintenance of the replicated source code copies becomes a major technical issue. Two sets of consistency maintenance techniques have been devised, one for each of the two types of editing operations and real-time collaboration sessions. To support consistency maintenance for real-time cluster-level editing operations, we derived a comprehensive set of conflicting operations with certain conditions, and specified the resolution results for all pairs of conflicting operations, which have been designed for preserving all conflicting operations' effects and preventing the loss of work as much as possible. To support consistency maintenance for real-time file-level editing operations, the existing OT technique and solution have been applied for maintaining multiple consistency properties including convergence and intention preservation.

Secondly, for supporting dynamic session membership and flexible switching among collaboration sessions and modes, a distributed join-protocol has been devised for accommodating new members at any time in an ongoing real-time collaboration session with other existing members. The protocol is capable of ensuring the consistency of source code in the face of dynamic session membership changes and concurrent editing, and achieving smooth transitions with minimum overheads and inconvenience to the end-users.

Thirdly, to preserve the compatibility with conventional non-real-time collaborative programming tools and environments, the NRTCoS component incorporates existing version control systems, and cooperates with other ATCoPE components in collectively supporting non-real-time collaboration functionalities in two aspects. For supporting conventional non-real-time collaborative programming, the system incorporates existing non-real-time collaboration services and provides the same working process, user interface and operation semantics as conventional IDEs with version control clients. For supporting any-time collaborative programming in terms of the interactions among non-real-time and real-time collaborating programmers/groups, ATCoPE bridges non-real-time collaboration services and real-time collaboration sessions by means of a Non-Real-Time Collaboration Proxy associated to each real-time collaboration session, which represents the real-time collaborating group as a single client of the NRTCoS component, and transmits version control operations and data between the NRTCoS component and the real-time collaboration session.

In the next chapter, we will address the semantic consistency issue in real-time collaborative programming with ATCoPE, and contribute a package of novel approaches, techniques and solutions for supporting semantic conflict prevention.

Chapter 4.

DAL: Dependency-based Automatic Locking for Supporting Semantic Conflict Prevention in ATCoPE

4.1 Introduction

One general requirement and major technical issue in collaborative editing systems is the consistency of the shared working artifacts. Generally, there are two classes of consistencies, namely *syntactic consistency* and *semantic consistency*. Syntactic consistency is mainly concerned with whether editing operations generated from multiple collaborating sites result in the same document state, whereas semantic consistency is concerned with whether the same document state is meaningful or correct in specific application domains [92][96]. The OT (Operational Transformation) technique discussed in Section 3.2.2 is sophisticated in supporting syntactic consistency maintenance, but not capable of dealing with semantic consistency maintenance [94].

Different collaborative systems may impose diverse semantic consistency requirements due to the different domain-specific meaning of the compatibility among concurrent work performed by collaborators. For example, in real-time collaborative text editing systems, semantic consistency may be concerned with the grammatical correctness of an English sentence, the logical compatibility among different paragraphs composed by different authors, and the compatibility among various ideas for writing the document. Particularly, in collaborative programming systems, semantic consistency is concerned with *whether the shared source code is correct with respect to programming language*

rules and problem-solving logic, and whether multiple source code modules modified and maintained by different programmers are compatible with each other.

The risk of semantic conflicts in collaborative programming depends on the nature of the collaborative work and the relationships among collaborative programming tasks by multiple programmers.

In non-real-time collaborative programming where collaborating programmers work on independent programming tasks, semantic conflicts among collaborative programming work are less likely to occur because the semantic consistency among various programming modules should have been maintained in software design activities (e.g., modularization, modeling) if designers have followed good design principles (e.g., increasing modularity, reducing coupling between modules, separation of concerns). For example, suppose that there are two isolated programming modules (two Java classes) named M_a and M_b assigned to programmers P_a and P_b respectively, and there is an invocation from a method in M_a to another method in M_b . In case that P_b makes some changes in the implementation of the invoked method, the changes should be transparent to P_a 's programming work without causing incompatibility among them, provided that the interface remains unchanged and both P_a and P_b have complied with the semantics of the related methods as specified in the design document.

Differently, in real-time collaborative programming, semantic inconsistencies are more likely to occur because concurrent real-time collaborative programming tasks on interdependent programming modules are more likely to produce source code with incompatible problem-solving logic. In addition, the risk of incompatible problem-solving logic (thus the semantic inconsistency) is usually higher as the scope of collaborative

programming modules/artifacts is smaller. For example, real-time collaborative programming tasks targeting the same source code file are more likely to produce semantic conflicts than those targeting different source code files.

One goal of this research is to support semantic consistency maintenance (or namely *semantic conflict prevention*) under the ATCoPE framework. In this research, we focus on semantic conflicts among real-time collaborative programming work in the same source code file (i.e., the same class in object-oriented programming) to investigate and develop core techniques for supporting semantic conflict prevention. However, techniques derived in this research are not limited to supporting semantic conflict prevention in the same source code file, but also applicable to supporting semantic conflict prevention in different source code files and modules as supported in ATCoPE.

Major research issues and solutions presented in this chapter are briefly introduced as follows. Firstly, to understand the natures of semantic conflicts in real-time collaborative programming, we explored three representative collaborative programming scenarios, presented the in-depth analysis, and derived the general conditions of semantic conflicts. Secondly, to deal with the semantic conflicts, we proposed and compared several alternative approaches, and contributed a dependency-based automatic locking (DAL) approach for supporting semantic conflict prevention. Thirdly, to address the challenge of the consistent and efficient dependency graph (DG) maintenance, we contributed an implicit derivation approach with supporting techniques. Fourthly, to realize the DAL approach, we devised the algorithms and procedures including DAL permission check and locking state update. Major contributions presented in this chapter include: (1) the exploration and in-depth analysis on representative collaborative programming scenarios; (2) the DAL approach for supporting semantic conflict prevention; (3) the formal

definitions of the basic region, the dependency relationship and the DG; (4) the conditions and algorithms for the DAL permission check; and (5) the implicit derivation approach and techniques for efficient maintenance of the DG and locking states.

The rest of this chapter is organized as follows. In Section 4.2, we use a classical programming task to illustrate representative semantic conflicts in real-time collaborative programming scenarios. In Section 4.3, we discuss and compare several alternative approaches to dealing with semantic conflicts, and propose a novel dependency-base automatic locking (DAL) approach for supporting semantic conflict prevention. In Section 4.4, we define several core concepts for the DAL approach, including basic region, dependency relationship and dependency graph. In Section 4.5, we present the DAL permission check, which is the key of the locking mechanism. In Section 4.6, we discuss core technical issues of the DAL approach, including dependency graph and locking state maintenance. Following the established technical approaches and solutions, the basic DAL scheme is presented in Section 4.7, which includes a permission check procedure, a locking state update procedure, integrated operation handlers, and DAL locking release mechanisms. Finally, in Section 4.8, we summarize this chapter.

4.2 Exploring Semantic Conflicts in Real-Time Collaborative Programming Scenarios

In this section, we use a classical programming task - *Stack* implementation - to illustrate representative semantic conflicts that may occur during real-time collaborative programming sessions. We implement the *Stack* as a Java class that stores integers as the stack elements and provides a set of simple stack functionalities, including: (1) *pushing* an element onto the top of the stack; (2) *popping* the top element out of the

stack; (3) *retrieving* the top element of the stack; and (4) *checking* whether the stack is empty or not.

Suppose that two collaborating programmers¹, denoted as P_1 and P_2 , are jointly conducting this programming task in a real-time collaboration session, where the syntactic consistency of the shared source code file is guaranteed by means of the OT technique² [95]. After a period of collaborative work, the source code evolves into the state as shown in Figure 4.1.

The constructor *Stack* takes a parameter that specifies the maximum size of the stack; the method *push* pushes an integer into the stack; the method *pop* pops the top element out of the stack; the method *top* returns the top element of the stack; and the method *isEmpty* tells whether the stack is empty or not.

The basic idea of the stack implementation is to maintain a linear integer array *int store[]* as the internal data structure for storing stack elements. In addition, the field *int max_length* indicates the maximum size of the stack (as specified by the constructor), and the field *int top* indicates the current index of the top element of the stack.

¹ Real-time collaboration sessions in case studies of this chapter involve only two programmers in order to simplify the illustrations, but all discussions also apply to programming scenarios with more programmers.

² The assumption that the syntactic consistency of the shared source code file is maintained by means of the OT technique applies to all examples and discussions in this chapter.


```

1 // Stack.java
2 package stack;
3
4 public class Stack {
5     protected int store[];
6     protected int max_length;
7     protected int top;
8
9     public Stack(int size) {
10         store = new int[size];
11         max_length = size - 1;
12         top = -1;
13     }
14
15     public boolean push(int num) {
16         if (top == max_length)
17             return false;
18         store[top] = num;
19         return true;
20     }
21
22     public boolean pop() {
23         if (top == -1)
24             return false;
25         top--;
26         return true;
27     }
28
29     public int top() {
30         return store[/*TODO*/];
31     }
32
33     public boolean isEmpty() {
34         return (top == -1);
35     }
36 }

```

Figure 4.1: Java source code of the *Stack* implementation

4.2.1 Case 1: Semantic Conflict between Concurrent Editing Operations in a Self-Contained Source Code Segment

As illustrated in Figure 4.2, P_1 and P_2 are concurrently editing the same method *push* at one moment. Initially, the source code segments at the two sites are the same and hence

syntactically consistent. However, according to the problem-solving logic, there exists a *programming error*: the *top* index should be increased by one before placing the pushed integer into the *store* array. Suppose that both P_1 and P_2 discover this error and attempt to fix it concurrently in different ways as follows:

Figure 4.2: Semantic conflict between concurrent editing operations in a self-contained source code segment

1. P_1 adds an increment operator to the left of *top* at line 18. It correctly fixes the error in P_1 's copy of the source code;
2. P_2 adds a standalone statement at line 18 to increase the value of *top*. It also correctly fixes the error in P_2 's copy of the source code.

After the two concurrent modifications are propagated to each other³, the two copies of the source code are identical and still syntactically consistent, as shown in Figure 4.2. Unfortunately, the syntactically consistent result remains semantically incorrect: there is still a *programming error* because the *top* index has been increased twice after the two concurrent modifications. In this case, each of them could fix the original programming error when executed individually, but the two modifications are logically (semantically) incompatible and the merged result converts the original programming error into a new one. In general, such kind of semantic conflicts may occur if multiple programmers are concurrently modifying the same self-contained source code segment (e.g., a method, a field) in semantically incompatible ways. Concurrent changes in a self-contained source code segment may produce programming errors (semantic conflicts) even if each individual change is error-free.

4.2.2 Case 2: Semantic Conflict between Concurrent Editing Operations in a Method and a Referenced Field

Different from *Case 1*, P_1 and P_2 are concurrently working in different segments of the same source code file, as shown in Figure 4.3:

1. P_1 focuses on the definition of the field *store*;

³ In this example, editing operations at each site are grouped into a single one for simplification. In real collaborative editing environments, editing operations are propagated more frequently (e.g., character-by-character) to achieve real-time notification. However, propagation frequency does not change the nature of semantic conflicts illustrated here. The same simplification will be applied to similar occasions in this chapter.

2. P_2 focuses on implementing the incomplete method *top*, which retrieves the top element of the stack.

Figure 4.3: Semantic conflict between concurrent editing operations in a method and a referenced field

Under such circumstances, one programmer, while focusing on his/her own work, may not pay attention to where and what others are working. Workspaces of the collaborators may not be adjacent to each other and could not be displayed in the same screen [89], bringing adverse impact on the awareness of the collaborative work. In this case, P_1 is working at line 5 while P_2 is working between lines 29-31, so they may not be able to view each other's workspace. At one moment, P_1 and P_2 are performing the following tasks concurrently:

1. P_1 is changing the internal data structure for the stack element storage (i.e., the field *store*) from an integer array into a *List* object provided by the standard Java library;
2. P_2 is completing the implementation of the method *top* based on the original definition (i.e., before the changes made by P_1) of the field *store*.

Clearly, the two concurrent changes are error-free when performed individually at each site, but their combined result becomes semantically incorrect: *a programming error* (i.e., a data type mismatch between the field *store* and the method *top*) has been introduced. The root of this problem is that P_1 and P_2 are concurrently changing source code segments with a *method-field-reference* dependency relationship: the method *top* depends on the field *store* because the former references the latter.

4.2.3 Case 3: Semantic Conflict between Concurrent Editing Operations in a Method and another Invoked Method

Suppose that an additional method, *void popList (int size)*, is added into the *Stack* class, which provides the convenient functionality of removing *size* consecutive elements at the top of the stack. The basic implementation approach is to invoke the existing *pop* method multiple times (determined by the *size* parameter) in a loop. As illustrated in Figure 4.4, the initial implementation of this new method has a problem: the method *popList* does not report any error message in case of invalid stack operations (e.g., the *pop* operation is issued when the stack is empty). Suppose that this problem is detected by both P_1 and P_2 and they start fixing the problem concurrently as follows:

1. P_2 is modifying the method *popList* by changing the return type of *popList* from *void* to *boolean*, checking the return value⁴ of the invoked method *pop*, and returning a *false* value if a *false* value has been returned from the invocation of the

⁴ The *pop* method reports the error message in the form of a *boolean* value so that the invoker could obtain whether the pop operation is successful or not by checking this return value.

pop method.

2. P_1 , who is working on the implementation of the method *pop*, decides to adopt the Java exception mechanism to substitute the *boolean* return value for reporting the error message: a Java exception object is created and thrown in case of an invalid stack operation issued.

Figure 4.4: Semantic conflict between concurrent editing operations in a method and another invoked method

This scenario is illustrated Figure 4.4. Once again, the concurrent changes are error-free when performed individually, but a *programming error* has been introduced: there is a

mismatch on the way of reporting invalid stack operation between the method *popList* and the method *pop*. The root of the problem is that the two programmers are concurrently changing different source code segments with a *method-method-invocation* dependency relationship: the method *popList* depends on the method *pop* because the former invokes the latter.

4.3 Dealing with Semantic Conflicts

4.3.1 General Conditions of Semantic Conflicts

In Section 4.2, we investigated three cases of semantic conflicts that may occur in real-time collaborative programming work. The three cases are not exhaustive, but representative in illustrating semantic conflicts in real-time collaborative programming sessions. From these cases, we can derive that semantic conflicts may occur under the following general and necessary conditions:

1. Multiple programmers are performing *concurrent editing operations*;
2. The concurrent editing operations are performed in the same *self-contained* source code segment or in multiple source code segments with *dependency relationships*;
3. The concurrent editing operations are *semantically incompatible*.

All illustrated cases meet these conditions. The third condition, *semantic incompatibility*, may take various forms. For example, semantic incompatibility may refer to *incompatible problem-solving logic*. *Case 1* is an example of *incompatible design of the interior logic of a self-contained source code segment*. In general, the interior logic within a self-contained source code segment (e.g., a method) is often ad hoc and unstructured

during the programming process. Concurrent editing within such source code segments may easily produce semantic conflicts. For another example, semantic incompatibility may refer to *incompatible usage of programming language rules* (e.g., language syntax and mechanism). In *Case 2*, collaborators have different ideas on the definition of the field *store*; in *Case 3*, collaborators adopt different error handling mechanisms.

4.3.2 General Approaches to Dealing with Semantic Conflicts

There are two general approaches to dealing with semantic conflicts: (1) *conflict resolution*, which allows conflicts to occur, and then detects and resolves them; and (2) *conflict prevention*, which prohibits conflicts from happening.

The *conflict resolution* approach is more suitable for dealing with conflicts that are easy to detect and cheap to resolve, whereas the *conflict prevention* approach is more suitable for conflicts that are difficult to detect and costly to resolve. In general, semantic conflicts are difficult to detect and resolve as they are often introduced by multiple programmers in concurrent work, which looks *error-free* from the perspectives of individual programmers. Moreover, semantic conflicts may result in *programming logic errors* that cannot be automatically detected by coding or compilation tools but have to be detected by programmers during source code review or testing, which is time-consuming and costly. Some logic errors may even not be detected by reviews or testing, and it is dangerous to deliver software products with such latent bugs. Therefore, the conflict prevention approach is preferable for dealing with semantic conflicts in real-time collaborative programming.

Given the three necessary conditions of semantic conflicts presented in Section 4.3.1, conflict prevention can be achieved by breaking any one of them:

1. *Preventing concurrent work*, which disallows multiple programmers to work in the same source code file concurrently;
2. *Preventing concurrent work in the same self-contained source code segment or in different source code segments with dependency relationships*, which allows concurrent work in the same source code file as long as such kind of work does not occur in the same self-contained source code segment or in source code segments with dependency relationships;
3. *Preventing incompatible work*, which allows programmers to work concurrently in any part of the source code file as long as they never produce incompatible work.

The first approach is the easiest to achieve (by simply locking the entire source code file for the unique programmer who is editing it), but too restrictive because it would eliminate the benefits of real-time collaboration. The third approach is most liberal in terms of maximum concurrency and flexibility, but infeasible to achieve because the system has no way to understand whether one source code segment is logically (semantically) compatible with another one. The second approach is most promising as it is able to support fine-grained conflict prevention while supporting concurrent work, and is technically achievable as well. Therefore, the approach of *preventing concurrent work in the same self-contained source code segment or in different source code segments with dependency relationships* is adopted to deal with semantic conflicts in real-time collaborative programming.

4.3.3 Dependency-based Automatic Locking (DAL)

Locking is a standard technique for mutual exclusion and conflict prevention [15]. To

achieve fine-grained conflict prevention in real-time collaborative programming, we propose a *dependency-based locking* approach: *a programmer is not allowed to edit a source code segment until obtaining exclusive locks (1) on the source code segment to be edited and (2) on those source code segments that are depended by the one to be edited.*

An important design issue with the dependency-based locking is how to place locks selectively on those source code segments with dependency relationships in a real-time collaborative programming session. One approach is the *manual locking*, which requires programmers to decide when and where to place/release locks. The programmer has to (1) manually specify the scope of the source code segment to be edited; (2) determine those depended source code segments and specify their scopes; (3) request all required locks from the system; and (4) wait until all requested locks are granted before continuing with the editing work. Upon completion of the editing work, the programmer has to manually release the locks. Such a process creates extra overheads to programmers and may cause them to lose focus on their major programming work. Furthermore, they may encounter difficulties in correctly analyzing the dependency relationships among source code segments, or fail to specify the correct scopes for the source code segments.

To overcome these problems in manual locking, we propose a *Dependency-based Automatic Locking* (DAL) approach, which requires the DAL system to automatically determine the scope of the source code segment to be edited and the scopes of the depended source code segments, grant locks on them, and release those locks when the editing work is completed. With automatic locking, programmers will be able to concentrate on their major programming work without having to care about locking.

4.4 Basic Concepts of the DAL Approach

Under the DAL approach, the self-contained source code segment is a basic element involved in the locking mechanism. This element and several related concepts are defined in this section.

Definition 4.1: Basic Region and Open Area

A *basic region* is a continuous sequence of source code that forms a semantically meaningful and self-contained segment. An *open area* is a continuous sequence of source code outside all basic regions.

Alphabetic symbols are used to represent basic regions. For example in Figure 4.5, each method/field of the *Stack* class is identified as a basic region, and other parts of the source code are regarded as open areas.

Definition 4.2: Dependency Relationship

For any two basic regions A and B , if A *depends on* B in terms of semantics (functionality, problem-solving logic and/or programming language rules), then there is a *dependency relationship* from A to B , denoted as $A \rightarrow B$, and B is called a *depended region* of A . If there exists no dependency relationship between A and B (neither $A \rightarrow B$ nor $B \rightarrow A$), then A and B are *independent*. Dependency relationship is *transitive*: given three basic regions A , B and C , if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

Dependency relationships may vary in different application domains and there may exist a variety of methods in modeling basic regions and dependency relationships. Without losing generality, we use a simple but representative dependency model to motivate and illustrate the design of basic mechanisms for the DAL approach, but the underlying

DAL mechanisms and techniques are generic and independent of specific dependency modeling methods. In this model, there are two types of basic regions - *method* and *field*, and two types of dependency relationships: (1) *method-field-reference* dependency relationship: if method M references field F , then $M \rightarrow F$; and (2) *method-method-invocation* dependency relationship: if method M_a invokes method M_b , then $M_a \rightarrow M_b$.

Figure 4.5: Basic region, dependency relationship, and dependency graph

All basic regions and dependency relationships embedded in a source code file can be represented by a *dependency graph* as follows.

Definition 4.3: Dependency Graph (DG)

A *dependency graph (DG)* is a directed graph, in which: (1) a *node* represents a *basic region* in the source code; and (2) an *arrow* from node *A* to node *B* represents a dependency relationship $A \rightarrow B$.

Based on the dependency model, all basic regions and dependency relationships for the *Stack* source code are identified and illustrated as a DG in the right part of Figure 4.5.

4.5 DAL Permission Check: Key of the Locking Approach and Mechanism

When a programmer is permitted to edit a source code region that is not locked by any programmer (regarded as a *free region*), the DAL system automatically places *exclusive locks* on this *working region*⁵ and its *depended regions*. Consequently, no one else can work in any of these locked regions until those locks are released. This basic DAL mechanism ensures the effectiveness of semantic conflict prevention by prohibiting concurrent access on selected source code regions (i.e., in the same source code region or in different source code regions with dependency relationships).

Under the basic DAL scheme, for each local editing operation to be performed on the

⁵ A *working region* refers to a basic region that the programmer is editing or attempts to edit.

source code, the *permission check* procedure is applied to examine and grant/deny the editing permission, according to the following *permission check conditions*.

Definition 4.4: DAL Permission Check Conditions

Permission to a local editing operation O is granted only if:

- 1) O 's location falls in an open area (free for all collaborating programmers to edit concurrently without requiring locks); or
- 2) O 's location falls in a basic region W , and:
 - a) W is already locked by this programmer (granted and recorded previously) as a *working region*; or
 - b) Neither W nor W 's depended region is locked by any other collaborating programmer.

To illustrate how the DAL permission check works, see the example in Figure 4.6 where the DG is extracted from Figure 4.5. If one of the collaborating programmers, namely P_1 , attempts to edit the method *pop*, which is a free region and its depended regions are free regions as well, then the editing is permitted according to Condition (2-b) in Definition 4.4. In the meantime, the DAL system automatically locks this region (denoted as basic region F) and its depended region *int top* (denoted as basic region C) for P_1 . Consequently, no other programmer can edit any of these regions (F and C). Furthermore, even if another programmer, namely P_2 , attempts to edit a source code region that is outside these locked regions such as the method *popList* (denoted as basic region I), the editing would also be denied by the DAL system because its depended regions, including the method *pop* (basic region F) and the field *int top* (basic region C), have already been exclusively locked for P_1 ; P_2 cannot obtain all required locks (on I , F and C) at that moment, and therefore has to wait until those locks held by P_1 are released.

Figure 4.6: Simple illustration of the DAL permission check

4.6 Dependency Graph and Locking State Maintenance

4.6.1 General Requirements

One fundamental issue in technically realizing the DAL approach is the DG maintenance, including the determination of the location and scope of each basic region in the source code, as well as the dependency relationships among them. The DG must always be *consistent* with the source code: for each method or field in the source code, there must be a corresponding basic region (with the correctly recorded location and scope of the basic region) in the DG, and vice versa; for each method-field-reference or method-method-invocation relationship in the source code, there must be a corresponding dependency relationship between corresponding basic regions in the DG, and vice versa. Moreover, because the source code contents are continuously updated during the programming process, the DG must be continuously updated accordingly as well. In addition, the interactive and dynamic nature of the working environment imposes a *real-time efficiency* requirement: the DAL system must support efficient and incremental DG updating in order to keep the DG staying consistent with the source code in real-time.

Another fundamental issue is the locking state maintenance, which is concerned with keeping track of which regions are locked by whom. The *locking state* is the key data structure for performing permission check: whenever a programmer issues a local editing operation on the source code, the recorded locking state is consulted to examine whether the programmer is permitted to work in the targeted source code region. It is not necessary for the locking state to cover all basic regions in the source code (i.e., covering the whole DG). Only those source code regions that are currently locked by

programmers should be recorded in the locking state in order to facilitate the DAL permission check. Similar to the DG, the locking state is also dynamic in nature, which may be changed from time to time as the source code structure (i.e., in terms of basic regions and dependency relationships) changes or any programmer's working region changes. Such changes performed on the locking state are regarded as the process of *locking state update*.

4.6.2 The Explicit Maintenance Approach

One possible approach to maintaining the DG and locking state is the *explicit maintenance* approach, which records and maintains the DG and locking state separately from the source code. An example of the explicit DG representation has been illustrated in the right part of Figure 4.5. Under the explicit maintenance approach, whenever a local editing operation is performed on the source code, corresponding update operations must be performed on the DG data structure immediately to maintain correct and consistent mapping between the DG and the source code. The locking state can be integrated with the DG by associating relevant DG nodes (that are locked by certain programmers) with locking information. Such explicit maintenance approach is intuitive to understand, but based on our technical experiments and analysis, it brings several major problems.

Firstly, the complexity involved in the consistency maintenance is overwhelming. As aforementioned, one technical requirement is the consistency between the DG and the source code, which can be named as the *DG-SC (dependency graph - source code) consistency*. In a distributed real-time collaborative programming session, the replication of the DG brings another consistency issue named as *DG-DG (dependency graph - dependency graph) consistency*: multiple replicas of a DG must be consistent over all col-

laborating sites in real-time, which is similar to the syntactic consistency among replicas of the shared documents in real-time collaborative editing systems [95]. Past research has shown that maintaining consistency among document replicas is a challenging task, which requires sophisticated techniques such as the OT technique. However, existing syntactic consistency maintenance techniques cannot be directly applied to DG replicas, as they have been commonly devised for consistency maintenance of the contents of the shared document in collaborative editing systems where editing operations are directly performed on the document contents [93]. Differently, for realizing the explicit maintenance approach, it is necessary to maintain the consistent mapping between the document contents and an additional document structure, and to maintain the consistency of the replicated document structures where editing operations are not directly performed on them but indirectly affecting them. It is absolutely a challenge to invent such an additional consistency maintenance technique.

Secondly, the space and time overheads for explicit maintenance can be high. The explicitly maintained DG, which is a separate graphical representation of the source code structure, can take substantial memory space, especially for large-size source code. Editing operations may create/delete/modify source code regions and change the dependency relationships among them, and therefore the explicit DG maintenance requires additional time overheads, including (1) the time overheads in detecting and analyzing the creations/deletions of source code regions; (2) the time overheads in detecting and analyzing the changes of the properties of source code regions (e.g., the starting position and length of each source code region); (3) the time overheads in detecting and analyzing the creations/deletions of dependency relationships; and (4) the time overheads in applying these update operations in the explicit DG data structures. Such time over-

heads can be substantial if an editing operation has impacts on multiple source code regions and the dependency relationships among them. In addition, some editing operations may also trigger the process of locking state update on the working region and its depended regions. It is a significant technical challenge to perform all these DG update and locking state update operations on a set of potentially large and complex DG data structures for every editing operation efficiently without causing noticeable latency.

4.6.3 The Implicit Derivation Approach

To overcome the problems involved in the explicit maintenance approach, we propose an *implicit derivation* approach. Under this approach, the DAL system does not maintain the DG separately, but relies on *dependency derivation techniques* to analyze and derive dependency relationships among specified regions within the source code file on demand and in real-time. It is *implicit* in the sense that the DG is implicitly embedded inside the source code. When a programmer issues an editing operation at a certain location of the source code, the dependency derivation technique is invoked to derive the working region of the editing operation and its depended regions, which are necessary in performing DAL permission check and locking state update.

The implicitly embedded DG is essentially a set of structural information of the source code, which is represented as an *abstract syntax tree* (AST)⁶ and maintained by underlying mechanisms of the programming environment. The implicit DG is automatically updated with the source code contents in real-time. By means of the source code struc-

⁶ http://en.wikipedia.org/wiki/Abstract_syntax_tree

tural information, the dependency derivation technique can be efficiently applied. Given any location within the source code, the dependency derivation technique is able to determine whether this location falls in a basic region (e.g., method/field) or an open area. If the location is within a basic region, the dependency derivation technique is able to retrieve the scope of this working region, as well as the locations and scopes of its depended regions. This basic derivation functionality can be recursively invoked to derive all depended regions with respect to a given working region in the source code.

Compared to the explicit maintenance approach proposed in Section 4.6.2, the implicit derivation approach has significant advantages. Firstly, it is able to achieve both DG-SC and DG-DG consistencies at the same time. The DG-SC inconsistency can be avoided because all dependency relationships among source code regions are always derived from the latest source code dynamically, and thus the DG-SC consistency is automatically guaranteed provided that the dependency derivation technique works properly and correctly. On the other hand, the DG-DG inconsistency can be eliminated for free because the consistency of the source code replicas has been guaranteed by syntactic consistency maintenance techniques such as OT, and thus the DG-DG consistency is automatically achieved provided that the same dependency derivation technique is applied at all collaborating sites.

Secondly, the implicit derivation approach is capable of avoiding most of the space and time overheads under the explicit maintenance approach. The space cost can be avoided simply because there is no separate DG maintained; and the time cost on updating the DG is avoided because an editing operation performed on the source code will automatically affect and update the embedded DG, and thus there is no need to perform additional DG updating operations.

Despite these advantages, the implicit derivation approach has several technical issues of its own. The first issue is the design and implementation of the dependency derivation technique that is able to analyze the source code regions and dependency relationships efficiently in real-time. In this research, we have achieved this functionality by invoking real-time source code syntax analysis functions that are commonly available in sophisticated programming systems, and the detailed techniques will be presented in Section 4.7.1. The second technical issue is the need for separate representation of the locking state. Under the implicit derivation approach, dependency relationships among source code regions are embedded inside the source code, but the locking state is external to the source code, and has to be separately maintained. In essence, the DAL locking state is actually a partial DG associated with locking information. Compared to maintaining the whole DG under the explicit maintenance approach, the cost of maintaining a partial DG is much lower. Furthermore, within this partial DG, the DAL system does not need to maintain the location and scope information for those locked regions and update them for every editing operation in real-time, because such location and scope information can be automatically derived from the latest source code by the dependency derivation technique. However, since the locking state is replicated (together with the source code replica) at all collaborating sites, consistency maintenance among multiple locking state replicas requires additional techniques to achieve.

4.6.4 DAL Locking State Data Structures

Under the implicit derivation approach for DG maintenance, the locking state is separately maintained outside the source code, which will be (1) consulted by the *permission check procedure* and (2) maintained by the *locking state update procedure*. The permission check procedure retrieves the information recorded in the locking state for exam-

ining the user’s editing permission on certain source code regions, and the locking state update procedure modifies the information in the locking state for reflecting the changes of locks granted on source code regions. The locking state data structures should be designed for accommodating efficient searching, retrieving and updating in order to facilitate fast permission check and locking state update.

Based on the DAL locking mechanism, the essential information to record in the locking state is which source code regions are locked by which users. As presented in Figure 4.7, a *DAL Table* is designed for recording all locks that are currently granted on source code regions, and each of the locks is represented by a corresponding *DAL Lock* in the *DAL Table*, which is expressed as $\langle \textit{Region Reference}, \textit{Owner ID}, \textit{Region Type} \rangle$. The *Region Reference* indicates the corresponding element (method/field) in the source code contents (uniquely identified) on which the lock is placed. The *Owner ID* indicates the holder of the lock, and the *Region Type* indicates whether the locked region is a working region (denoted as *W*) or a depended region (denoted as *D*) with respect to the owner.

Figure 4.7: DAL locking state data structures

The right part of Figure 4.7 illustrates an instance of the *DAL Table* data structures when two programmers P_1 and P_2 are editing basic regions B and D respectively in a real-time collaborative programming session. As presented in the DG, four source code regions $\{B, C, D, E\}$ are currently locked by the programmers: working region B and depended region C are locked for P_1 , whereas working region D and depended region E are locked for P_2 . These four source code regions are currently involved in the locking mechanism, and thus there are four corresponding *DAL Locks* within the *DAL Table*. As illustrated in Figure 4.7 and mentioned in Section 4.6.3, the DAL locking state is essentially a partial DG, which does not cover source code regions that are not locked.

4.7 The Basic DAL Scheme

4.7.1 DAL Permission Check Procedure

Based on the permission check conditions defined in Definition 4.4 and the DAL locking state data structures presented in Section 4.6.4, the algorithm for the DAL permission check procedure can be designed accordingly. Several utility functions are designed for supporting the permission check procedure:

Table 4.1: Utility functions for DAL permission check procedure

Utility Function	Description
<i>GetWorkingRegion(O)</i>	To derive the working region of a given editing operation O . It returns a <i>Region Reference</i> (which points to the corresponding element in the source code contents) if O falls in the scope of a basic region, or returns <i>NULL</i> if O falls in an open area.

<i>GetDepRegionSet(W)</i>	To derive the set of all depended regions that the working region <i>W</i> depends on.
<i>CheckOwnership(R, U, S, T)</i>	To check whether a given source code region <i>R</i> is locked by the user <i>U</i> or by others (specified by <i>S</i> = <i>SELF/OTHERS</i>) as a working region or depended region, or without regarding the region type (specified by <i>T</i> = <i>WORKING/DEPENDED/ALL</i>).

The *GetWorkingRegion* function firstly retrieves the list of basic regions (i.e., fields and methods) within the current source code unit (i.e., class), and the retrieved information contains detailed properties of each basic region, including the positional parameters such as the *starting position* and the *length*. It then compares the positional parameters of the editing operation *O* with those of each basic region, and judges whether the editing operation falls in any basic region or not. The complexity of *GetWorkingRegion* is $O(N)$ where N is the amount of basic regions within the source code unit. The runtime overheads of *GetWorkingRegion* consist of the time spent on retrieving source code regions' information (denoted as T_1) and the time spent on searching for the working region (if any) within the retrieved list (denoted as T_2). Under the implicit derivation approach, all structural information of the source code is implicitly represented as an *abstract syntax tree* (AST), which is being updated with the source code contents in real-time by means of the IDE's real-time source code syntax analysis mechanisms. Therefore, T_1 is relatively short as the source code structural information is available at any time, which can be efficiently retrieved on demand. The overall time cost of this function thus approximates to T_2 , which mainly depends on the amount of basic regions.

The *GetDepRegionSet* function starts from an empty *depended region set* (DRS). If the

working region is a field (which has no depended region), an empty DRS is returned. Differently, if the working region is a method, all sub-elements within the method body (i.e., sub-nodes under the AST node that represents the method) are visited, and all field references and method invocations will be retrieved. If any referenced field or invoked method belongs to the same source code unit (i.e., class), it is then added into the DRS (provided that it has not been added into the DRS yet). If the newly added depended region is a method, the same operations will be applied to further derive the depended regions of this method. Such procedure is recursively applied until no more regions can be added into the DRS. In the worst case, a working region may depend on all other methods within the same class, and thus all field references and method invocations within the class are visited and analyzed. For each field reference or method invocation that is being visited, there are two essential operations performed: (1) to check whether the referenced field or invoked method is a basic region of the current source code unit (i.e., class); and (2) if so, to check whether it already exists in the DRS. It can therefore be derived that the complexity of *GetDepRegionSet* is $O(NM)$ where N is the amount of basic regions (i.e., methods and fields) and M is total amount of field references and method invocations in all methods. The runtime overheads of *GetDepRegionSet* consist of the time spent on retrieving AST nodes (denoted as T_1) and the time spent on applying the two essential operations to field references and method invocations being visited (denoted as T_2). Because the implicitly embedded AST has been continuously updated with the source code contents in real-time by the IDE, T_1 is relatively short as all AST nodes can be retrieved efficiently on demand. Therefore, the overall time cost of the function approximates to T_2 , which mainly depends on the amount of basic regions, the amount of depended regions in the DRS during the progress of function execution, and

the total amount of field references and method invocations within the working region and its depended regions.

The *CheckOwnership* function searches for the particular *DAL Lock* that relates to the specified source code region *R* by comparing *R* with the *Region Reference* of each *DAL Lock* in the *DAL Table*. Once the particular *DAL Lock* is found in the table, its properties (*Owner ID* and *Region Type*) are further examined to judge the ownership with respect to the specified conditions. Accordingly, the complexity of the *CheckOwnership* function is $O(K)$ where K is the amount of *DAL Locks* within the *DAL Table*. The runtime overheads of *CheckOwnership* consist of the time spent on searching for the *DAL Lock* (denoted as T_1) and the time spent on retrieving the properties of the *DAL Lock* (denoted as T_2). The overall time cost of *CheckOwnership* approximates to T_1 , as the process of reading the properties of a specified *DAL Lock* is straightforward and efficient.

Algorithm 4.1 presents the permission check procedure *PermissionCheck*(*O*, *U*) for examining the editing permission for a given local editing operation *O* issued by the local user *U*. If *O* falls in an open area, *PERMIT_OA* is returned to indicate that the permission is granted for an editing operation in an open area (no lock is required); if *O* falls in a basic region, the procedure further examines the locking states of the working region and its depended regions. If the targeted working region *W* is found being a working region that is already locked by the local user *U*, *PERMIT_WR* is returned to indicate that the permission is granted for an existing working region. If *W* is found being locked by other users, *REJECT* is returned to deny the editing permission. If no *DAL Lock* on *W* is found in the *DAL Table*, it must be a free region, and *W*'s depended regions are further derived and examined. If any of the depended regions is found being locked by other users, *REJECT* is returned to deny the editing permission; otherwise,

PERMIT_FR is returned to grant the permission for a free region (corresponding locks on working region and depended regions will be granted later by the locking state update procedure as presented in Section 4.7.2).

Algorithm 4.1: *PermissionCheck*(*O*, *U*)

```

W := GetWorkingRegion(O);
if W = NULL
    return PERMIT_OA;
if CheckOwnership(W, U, SELF, WORKING) = TRUE
    return PERMIT_WR;
if CheckOwnership(W, U, OTHERS, ALL) = TRUE
    return REJECT;
DRS := GetDepRegionSet(W);
for each Basic_Region D in DRS {
    if CheckOwnership(D, U, OTHERS, ALL) = TRUE
        return REJECT;
}
return PERMIT_FR;

```

As presented, the *PermissionCheck* procedure invokes *GetWorkingRegion*, *GetDepRegionSet* and *CheckOwnership* utility functions. Suppose that N is the amount of basic regions, M is the total number of field references and method invocations, and K is the amount of *DAL Locks* within the *DAL Table*. As derived, the complexity of *GetWorkingRegion* is $O(N)$, the complexity of *CheckOwnership* is $O(K)$, and the complexity of *GetDepRegionSet* is $O(NM)$. In addition, the complexity of the loop for checking the locking states of depended regions is $O(K * |DRS|)$ where $|DRS|$ denotes the amount of depended regions in the *DRS*. The overall complexity of *PermissionCheck* is therefore $O(N + 2K + NM + K * |DRS|)$. In the worst case, K may equal to N as all basic regions

may be involved in the *DAL Table*, and $|DRS|$ may approximate to N as a working region may depend on all other methods and fields, where $0 \leq K \leq N$ and $0 \leq |DRS| \leq N - 1$. The overall complexity of *PermissionCheck* can therefore be expressed as $O(N + 2N + NM + N * N)$, which can be simplified as $O(NM + N^2)$.

4.7.2 DAL Locking State Update Procedure

Based on the locking state data structures presented in Section 4.6.4, the algorithm for the locking state update procedure has been designed accordingly. Two utility functions are first designed for the locking state update:

Table 4.2: Utility functions for DAL locking state update procedure

Utility Function	Description
<i>GrantLocks</i> (W, DRS, U)	To place locks on working region W and depended region set DRS for user U in the <i>DAL Table</i> .
<i>ReleaseLocks</i> (U)	To remove all existing locks held by user U from the <i>DAL Table</i> .

The *GrantLocks* function inserts new entries (i.e., *DAL Locks*) into the *DAL Table* in accordance with the specified working region W and depended regions in the DRS . The complexity of *GrantLocks* is $O(1 + |DRS|)$ where $|DRS|$ denotes the amount of depended regions in the DRS . In the worst case, $1 + |DRS| = N$ when a working region depends on all other methods and fields in the class. Therefore, the complexity of *GrantLocks* can also be expressed as $O(N)$ where N is the amount of basic regions in the source code. The runtime overheads of the function completely depend on the size of the DRS , as the function executes the entry insertion operation for exactly $(1 + |DRS|)$ times.

The *ReleaseLocks* function visits all *DAL Locks* within the *DAL Table*. For each *DAL Lock* that is being visited, the function examines its properties and judges whether it is granted to the specified user *U*. The complexity of *ReleaseLocks* is $O(K)$ where K is the amount of *DAL Locks* within the *DAL Table*. In the worst case, all fields and methods of a class are locked, and the size of the *DAL Table* equals to the amount of basic regions in the source code. Therefore, the complexity of *ReleaseLocks* can also be expressed as $O(N)$ where N is the amount of basic regions in the source code. The runtime overheads of the function completely depend on the amount of *DAL Locks* within the *DAL Table*, as the function executes the entry examination operation for exactly K times.

Whenever an editing operation triggers any change of the locking state, the above two *locking operations* will be invoked to perform the locking state update. For instance, when a programmer starts the editing work in a free region, this region (as the working region) and its depended regions will be locked for this programmer. In another case where the programmer switches the workspace from a locked working region to another free region, the existing locks held by this programmer will be released first, and then a set of new locks on the new working region and depended regions will be granted to this programmer.

In addition, for each locking operation triggered by a local editing operation issued at a site, besides its execution on the local locking state, it is also necessary to propagate the locking operation to all remote sites for execution in order to achieve consistent locking state update over all collaborating sites. Under the basic DAL scheme, this consistent locking state update effect can be achieved by the following measures:

1. At a local site, whenever the locking state update procedure grants or releases a set

of locks with respect to an editing operation, a locking operation tag *LOCK* or *RELEASE* will be additionally attached to the editing operation and then propagated together with the editing operation to all remote sites;

2. At a remote site, whenever a remote editing operation arrives with a locking operation tag (*LOCK* or *RELEASE*) attached, the locking state update procedure will be performed accordingly, which updates the locking state consistently as the operation's local site behaves.

Algorithm 4.2 presents the procedure *LSUpdate*(*S*, *R*, *U*, *O*) for updating the locking state with respect to an editing operation *O* issued by the user *U*. The parameter *S* specifies whether the update is related to a local editing operation (specified by *S* = *LOCAL*) or a remote editing operation (specified by *S* = *REMOTE*). The parameter *R* passes the returned result of the permission check procedure for this editing operation to the locking state update procedure, if the editing operation is a local one.

For a local editing operation *O*, the locking state update procedure acts according to the result of the permission check: (1) if *O* is permitted for an open area, all existing locks held by *U* are released, and no lock is required for the editing in an open area; (2) if *O* is permitted for a free region, all existing locks held by *U* are released, and the new set of required locks are derived and granted; (3) if *O* is permitted for the existing locked working region, no locking state update action is needed.

For a remote editing operation *O*, the locking state update procedure acts according to the locking operation tag attached to the editing operation. If the *RELEASE* tag is attached, all existing locks held by the remote user *U* are released. If the *LOCK* tag is attached, the working region (at the location of the remote editing operation *O*) and its

depended regions are derived, and locks are granted on these involved regions to the remote user U .

Algorithm 4.2: $LSUpdate(S, R, U, O)$

```

if S = LOCAL {
    if R = PERMIT_OA {
        ReleaseLocks(U);
        AttachTag(O, RELEASE);
    }
    else if R = PERMIT_FR {
        ReleaseLocks(U);
        AttachTag(O, RELEASE);
        W := GetWorkingRegion(O);
        DRS := GetDepRegionSet(W);
        GrantLocks(W, DRS, U);
        AttachTag(O, LOCK);
    }
    else if R = PERMIT_WR
        // Do nothing
}
else if S = REMOTE {
    if CheckTag(O, RELEASE) = TRUE
        ReleaseLocks(U);
    if CheckTag(O, LOCK) = TRUE {
        W := GetWorkingRegion(O);
        DRS := GetDepRegionSet(W);
        GrantLocks(W, DRS, U);
    }
}

```

As presented, the $LSUpdate$ procedure invokes $GetWorkingRegion$, $GetDepRegionSet$,

GrantLocks and *ReleaseLocks* utility functions. Suppose that N is the amount of basic regions and M is the total amount of field references and method invocations in the source code. As derived, the complexity of *GetDepRegionSet* is $O(NM)$ whereas the complexities of the other three functions are $O(N)$. Since there is no loop involved in the algorithm, the overall complexity of *LSUpdate* is $O(NM)$.

4.7.3 Integrated Operation Handlers for Processing Editing and Locking Operations

Algorithm 4.3 presents the local operation handler $LOH(O, U)$ for processing a local editing operation O issued by the local user U . The handler first invokes the permission check procedure to examine the editing permission. If the permission is denied, O is prohibited from execution, and U is notified of the failure; otherwise, it is executed on the source code, and the *LSUpdate* procedure is then invoked to update the locking state accordingly. Finally, the operation is propagated to remote sites for execution.

Algorithm 4.3: $LOH(O, U)$

```

result := PermissionCheck(O, U);
if result = REJECT {
    Reject Execution of O and Notify User U;
    return;
}
Execute(O);
LSUpdate(LOCAL, result, U, O);
Propagate(O);

```

Algorithm 4.4 presents the remote operation handler $ROH(O, U)$ for processing a re-

remote editing operation O issued by a remote user U . O is first executed without permission check because O has been permitted at its local site and its permission at a remote site is guaranteed by nature. After the execution of O on the source code, the locking state is updated accordingly if necessary.

Algorithm 4.4: $ROH(O, U)$

```
Execute (O) ;  
LSUpdate (REMOTE, NULL, U, O) ;
```

In addition, it can be highlighted that the DAL mechanism is *deadlock-free*. Deadlock among multiple users (at collaborating sites) is impossible because the permission check procedure is performed in a non-blocking manner, thus eliminating the possibility of the *hold-and-wait* condition, which is a necessary condition for any deadlock. The permission check procedure always returns with either a success (i.e., the editing is permitted and locks are updated accordingly if necessary) or a failure (i.e., the editing is denied). In case that the editing permission is denied, the user is notified of the result but not blocked, and therefore s/he can temporarily continue the programming work in other parts (e.g., free regions or open areas) of the source code and later switch the workspace to this region again if the previously placed locks have been released.

4.7.4 DAL Locking Release Mechanisms

As specified in the locking state update procedure, whenever a programmer switches the workspace (editing location) from one source code region to another source code region or an open area, all locks granted on the original working region and depended regions will be automatically released by the DAL locking state update. By this measure, it is

ensured that at any time of the programming process, locks granted for a programmer always correctly cover the latest working region (if not in an open area) and depended regions, and never cover source code regions that are outside the current working region and depended regions. In addition to this automatic locking release mechanism, two more mechanisms for the DAL locking release are designed as follows.

Firstly, locks granted for any programmer will be automatically released if the programmer has not generated any operation in the programming environment (e.g., text typing, cursor moving, page scrolling) for a duration of certain length (e.g., 5 minutes). This mechanism ensures that locks granted to a programmer are effectively used by the programmer, and eliminates unnecessary holding of the DAL locks. For example, it can eliminate the situation that a programmer leaves the workstation and keeps away for a long time while other programmers cannot work on some of the source code regions due to locks held by the programmer who has left already. In addition, within the user interface, a countdown timer will be displayed to indicate the remaining time when it is near the end of the time-out duration for locking release. In this case, if the programmer still wishes to continue the programming work in the current workspace, s/he can simply move the mouse to prevent the locks from being released. This interface feature is useful for the situation that a programmer seems to be inactive on the programming work but actually keeps thinking on some programming issues. To achieve flexibility, the length of this time-out duration can be adjusted by the programmer, and may take a default value (such as 5 or 10 minutes) pre-set by the DAL system.

The time-out-based automatic locking release mechanism can be realized by an internal timer within the DAL system. The timer is reset to the initial value (i.e., the time-out duration) whenever there is an input operation detected in the user interface, and keeps

running at all times if there is no operation generated by the user. When the remaining time reaches a certain value, the abovementioned countdown timer will be displayed at the user interface; when the remaining time reaches zero, the *ReleaseLocks* function designed in Section 4.7.2 will be executed accordingly to remove all locks held by this programmer. In addition, to achieve consistent locking release over all collaborating sites, whenever the DAL system automatically releases a set of locks, if the locking release is not triggered by the locking state update procedure (and hence there is no editing operation to carry the *RELEASE* tag), a standalone message will be propagated to all other collaborating sites for executing the *ReleaseLocks* function.

Secondly, in addition to the above automatic locking release mechanism, a programmer may also manually release the locks by simply clicking a button provided in the user interface. This manual locking release feature is particularly useful for the circumstance where a programmer has completed the programming work in the current working region (with locks granted), but has no decision yet on which part to work in for the next step. By means of manual locking release, a programmer can make source code regions available for other programmers to edit as early as possible. Technically, the underlying procedures for this manual locking release feature are almost the same as the automatic locking release as presented above. Whenever a manual locking release operation is issued by the programmer, the *ReleaseLocks* function is executed, and a communication message is propagated to all remote sites for executing the same procedure.

4.8 Summary

In this chapter, we contributed a package of novel techniques for supporting semantic conflict prevention in real-time collaborative programming. By analyzing a set of rep-

representative programming scenarios for understanding the nature and general conditions of semantic conflicts, it has been derived that semantic conflicts may occur when multiple programmers are performing concurrent and incompatible editing work in the same source code region or in different source code regions with dependency relationships. Based on the analysis, a *Dependency-based Automatic Locking* (DAL) approach is proposed, and major technical issues and solutions for supporting the DAL approach are discussed and devised, including conditions and algorithms for the DAL permission check, implicit derivation approach and techniques for dependency graph and locking state maintenance, the design of the locking state data structures, algorithms and techniques for the DAL locking state update, integrated handlers for processing editing and locking operations under the basic DAL scheme, and the DAL locking release mechanisms. The novelty of the DAL approach lies in its capabilities of supporting automatic, responsive and fine-grained locking on selected source code regions and balancing conflict prevention, concurrent work and programmer convenience in real-time collaborative programming.

This chapter has focused on the essence of the DAL approach, devised the basic DAL scheme, and contributed core techniques under two constraints: (1) there exists no *regional overlapping relationship* among *concurrent locking operations*, in the sense that concurrent locking operations (triggered by concurrent editing operations) never result in multiple locks granted on the same source code region; and (2) the DG structure of the source code remains *static* during real-time collaborative programming sessions, in the sense that editing operations may change the textual contents of the source code, but never result in DG structural changes such as creation/deletion of basic regions and dependency relationships.

In the next chapter, we will extend the basic DAL scheme to allow collaborating programmers to (1) *concurrently* edit source code regions with *overlapping locking scopes* and/or (2) *dynamically* change the DG structure of the source code in real-time collaborative programming with semantic conflict prevention.

Chapter 5.

Extended DAL for Supporting Semantic Conflict Prevention in Unconstrained Real-Time Collaborative Programming

5.1 Introduction

For supporting semantic conflict prevention in real-time collaborative programming, we have proposed a novel dependency-based automatic locking (DAL) approach to automatically grant locks on selected source code regions with dependency relationships. The motivation of the DAL approach, core elements and concepts under the DAL framework, and the basic DAL scheme including permission check and locking state update procedures have been presented in Chapter 4.

Under the basic DAL scheme, for each local editing operation to be performed on the source code, the permission check is applied to grant or deny the editing permission, according to the *permission check conditions* specified in Definition 4.4. The basic DAL scheme enforces exclusive locking among programmers (i.e., no lock can be shared among multiple programmers), and the locking mechanisms work correctly under the following two general constraints:

1. There exists no *regional overlapping relationship* among *concurrent locking operations*, which prohibits multiple programmers from working concurrently in the following source code regions:
 - a) in the same working region;

- b) in different working regions with dependency relationships; or
 - c) in different independent working regions with common depended region(s).
2. The DG structure of the source code remains *static* during the collaborative programming session, which assumes that editing operations may change the textual contents of the source code, but never result in structural changes of the DG (i.e., creation/deletion of basic regions and dependency relationships).

In practice, these constraints under the basic DAL scheme are too restrictive for supporting flexible and unconstrained real-time collaborative programming. Based on the core techniques of the DAL approach as devised in Chapter 4, we present major extensions to the basic DAL scheme to allow collaborating programmers to *concurrently* edit source code regions with *overlapping locking scopes* and/or *dynamically* change the DG structure of the source code during real-time collaborative programming work.

Major research issues and solutions presented in this chapter are briefly introduced as follows. Firstly, to address the three major restrictions under the basic DAL scheme, we contributed a shared-locking scheme, which allows three types of shared-locking under well-defined circumstances based on a set of common design rationales. Secondly, to accommodate the shared-locking, we redefined the DAL permission check conditions, redesigned the DAL locking state data structures, and revised the three utility functions related to DAL locking states. Thirdly, to address the issue that an editing operation may affect both the operation-actor and non-operation-actors' locking states under the extended DAL scheme, we contributed a contextualization and full derivation (CFD) scheme with related techniques for supporting locking state update. Major contributions presented in this chapter include: (1) the shared-locking scheme which serves as the

cornerstone of the extended DAL scheme; and (2) the CFD locking state update scheme and techniques which have realized the extended DAL scheme.

The rest of this chapter is organized as follows. In Section 5.2, three major restrictions of the basic DAL scheme in supporting unconstrained real-time collaborative programming are analyzed. A novel shared-locking scheme for addressing the three restrictions altogether in a uniform framework is proposed in Section 5.3. Major technical issues and solutions for the extended DAL scheme are discussed and devised in Section 5.4, including the extended DAL locking state data structures, the contextualization and full derivation (CFD) scheme, and the correctness verification of the CFD locking state update scheme with respect to a set of correctness criteria. Comparisons between the DAL scheme and other locking schemes in related work are presented in Section 5.5.

5.2 Restrictions of the Basic DAL Scheme in Supporting Unconstrained Real-Time Collaborative Programming

The basic DAL scheme has three major restrictions in supporting unconstrained real-time collaborative programming, which will be analyzed in this section. The solution to these issues will be presented in the next section.

5.2.1 Restriction in Supporting Concurrent Work on Independent Working Regions with Common Depended Regions

Under the basic DAL scheme, when a programmer is permitted to edit a free region, the DAL system automatically places *exclusive locks* on this working region and all its depended regions. Consequently, no one else can work in any of the locked regions or in

any other region whose depended region has been locked, until those exclusive locks are released. The basic DAL scheme is too restrictive in supporting concurrent programming work, which can be illustrated by the example in Figure 5.1¹, where independent source code regions A and D have common depended regions B and C . Suppose that a programmer P_1 first started working in source code region A , and automatically obtained exclusive locks on working region A and depended regions B and C . During the period of P_1 's working in region A , another programmer P_2 attempts to work in region D , which is independent of P_1 's working region A (i.e., neither $A \rightarrow D$ nor $D \rightarrow A$).

Figure 5.1: Restriction in supporting concurrent work on independent working regions with common depended regions

¹ In this chapter, locks placed on source code regions are represented by tags associated to nodes in the DG: a lock granted to programmer P_x on source code region R is represented by a tag P_x placed beside the region R (i.e., node R in the DG), and a superscript (^W) is placed to further indicate that R is a *working region* of P_x , while there is no superscript if it is a *depended region* of P_x .

According to the general conditions of semantic conflicts (in Section 4.3.1), concurrent work on independent working regions A and D has no risk of semantic conflicts because they are independent of each other. However, the basic DAL scheme prohibits this kind of concurrent work due to the common depended regions B and C . The DAL system justifies that P_2 should obtain exclusive locks on working region D and depended regions B and C (as enforced by the permission check conditions in Definition 4.4), but cannot grant all required locks because depended regions B and C are currently exclusively locked by P_1 .

In real-world programming scenarios, such a circumstance is common, as class fields (or global variables in other languages such as C++) and utility methods are often referenced or invoked by multiple independent methods in the same class. However, under the basic DAL scheme, if one programmer is working on one of those methods, no one else is allowed to work on other independent methods that invoke the locked method, which is overly restrictive in supporting concurrent programming work and unnecessary for semantic conflict prevention.

5.2.2 Restriction in Supporting Concurrent Editing Operations with Overlapping Locking Scopes

In real-time collaborative programming environments over high latency communication networks such as the Internet, high local responsiveness is an essential requirement, which has led to the *replicated architecture* for the shared source code file, the DG, and the locking state data structures over multiple collaborating sites. Under the basic DAL scheme, whenever a programmer issues an editing operation on any part of the source code at the local site, the permission check is performed to examine whether the pro-

programmer is allowed to work at the targeted location of source code according to Definition 4.4, which consults the local locking state. To ensure non-blocking and responsive permission check, the procedure is executed locally without communication with remote sites (i.e., by consulting the local locking state only). With quick permission check, the user's local editing can immediately take effect if the permission is granted or be rejected if the permission is denied. At the local site, whenever a locking operation (i.e., grant/release locks) is triggered by a local editing operation, it is also propagated to all remote sites for execution together with the remote execution (replay) of the editing operation, so as to produce consistent locking states over all collaborating sites.

Under this fully distributed and responsive locking scheme, concurrent locking operations may conflict with each other. Multiple programmers may concurrently start working in the same free region or in different free regions with dependency relationships or common depended regions. If no special measure is taken, the free region will be locked by different programmers at different collaborating sites because all of the programmers can be granted the editing permission based on the information in their local locking state replicas. Such locking operations, which result in inconsistent locking states at different collaborating sites, are regarded as *conflicting operations*.

For example in Figure 5.2, if both P_1 and P_2 start to edit the same source code region A , both of them will be granted the editing permission because the permission check is solely based on the local locking state, and the targeted working region and depended regions are all free regions at both sites at the time of permission check. However, when the editing operations and the associated locking operations arrive at the remote sites, a locking operation conflict is produced: when P_2 's editing operation arrives at P_1 's site, the execution of the remote locking operation fails because the targeted source code re-

gions have already been locked for P_1 ; similarly, when P_1 's editing operation arrives at P_2 's site, the execution of the remote locking operation fails due to the same reason.

Figure 5.2: Locking operation conflict caused by concurrent editing operations

5.2.3 Restriction in Supporting Dynamic DG Editing Operations

All editing operations performed on the source code will change the textual contents of the source code, but textual changes may not necessarily lead to structural changes of the DG. For example, an editing operation may insert/delete several statements inside an existing source code region such as a method, which only expands/shrinks the scope of the basic region but does not create or destroy any basic region or dependency relationship, thus preserving the static DG structure.

However, the textual change effects of an editing operation sometimes may lead to the structural change of the DG. As illustrated in Figure 5.3, an editing operation may lead to the creation or deletion of a method (see Figure 5.3(i)), an invocation from a method

to another method (see Figure 5.3(ii)), or a reference from a method to a field (see Figure 5.3(iii)), thus *dynamically* changing the DG structure.

Figure 5.3: Dynamic DG editing operations

Generally, there are two types of *basic* DG changes: (1) creation or deletion of a basic region; and (2) creation or deletion of a dependency relationship. These basic DG changes cover all possible cases of dynamic DG changes caused by editing operations. In a real-world programming process, a single editing operation may cause *compound* DG changes that contain multiple basic DG changes. For instance, cutting and pasting a large piece of source code contents may have the effects of deleting/creating multiple source code regions and/or dependency relationships simultaneously, resulting in complex changes of the DG structure. In this thesis, we use the term *dynamic DG editing operation* to refer to those operations with the effect of changing the DG structure, and the term *static DG editing operation* for those without such effect.

In recognizing the effects of editing operations on the DG structure, one question arises: how to continuously preserve the effectiveness of semantic conflict prevention in the presence of dynamic DG changes during a programming process? Based on the principle of the DAL approach, one reasonable strategy is to continuously monitor the im-

pacts of editing operations on the DG structure, and dynamically adjust the locking state to ensure that the locks granted to each programmer always correctly cover the current working region and depended regions based on the latest source code, at any instant of time. For example in Figure 5.3(ii), after creating the method invocation from *method_A* to *method_B*, the locks granted to this programmer should be dynamically updated from $\{A^W\}$ to $\{A^W, B\}$ in order to reflect the newly created dependency relationship $A \rightarrow B$.

However, such dynamic locking state update may also cause locking operation conflicts, regardless of whether the dynamic DG editing operation is concurrent with another operation or not. For example in Figure 5.4, P_1 and P_2 are initially editing source code region A and C respectively, with corresponding locks granted to them. At one moment, P_1 creates a new dependency relationship $A \rightarrow C$. Consequently, P_1 's locking scope should be updated to cover the new depended regions C and D , which overlaps with the working region and depended region of P_2 , thus producing a locking operation conflict between P_1 and P_2 . In this simple case, no concurrency is involved as the editing operation happens sequentially after all previous editing and locking operations.

The situation may become more complicated in combination with concurrency, such as the case illustrated in Figure 5.5, where P_1 and P_2 are initially working in the same open area. P_1 is inserting the left brace, and P_2 is inserting the right brace concurrently. After the two concurrent editing operations have been executed at remote sites, a valid source code region is formed, and both of their latest editing locations fall into the same source code region A . By dynamic locking state update, both programmers' locking scopes should be updated to cover the newly created region A , thus causing a locking operation conflict between them.

Figure 5.4: Locking operation conflict caused by dynamic DG editing operations without concurrency involved

Figure 5.5: Locking operation conflict caused by dynamic DG editing operations in combination with concurrency

5.3 Shared-Locking for Addressing Restrictions of the Basic DAL Scheme

To address all three restrictions of the basic DAL scheme presented in Section 5.2, we propose a *shared-locking* scheme, which allows multiple programmers to share locks on overlapping source code regions under well-defined circumstances, while the effectiveness of the semantic conflict prevention is preserved. Under a uniform shared-locking scheme, the three restrictions can be solved altogether with consistent rationales, which will be elaborated and explained in the following three sub-sections respectively.

5.3.1 Shared-Locking on Common Depended Regions

We extend the basic DAL scheme to allow *shared-locking* on *common depended regions*. As illustrated in Figure 5.6 which reuses the case in Figure 5.1, under the shared-locking scheme, P_2 will be granted the editing permission on independent region D concurrently with P_1 's working on source code region A , holding an exclusive-lock on the working region D and shared-locks on depended regions B and C .

Figure 5.6: Shared-locking on common depended regions

Under such circumstance, the lock-sharing occurs among depended regions of multiple programmers (regions B and C in this example), which can be regarded as the overlapping of depended regions, denoted as *D-D Overlapping*.

As argued in Section 5.2.1, allowing concurrent work in independent working regions with common depended regions has no risk of semantic conflicts, but eliminates unnecessary restrictions on the concurrency of collaborative work. With this extension, when a programmer attempts to edit the source code, the editing permission is examined according to the *revised permission check conditions* in Definition 5.1 below, which allow shared-locking on common depended regions.

Definition 5.1: Revised Permission Check Conditions

Permission to a local editing operation O is granted only if:

1. O 's location falls in an *open area* (free for all collaborating programmers to edit concurrently without requiring locks); or
2. O 's location falls in a *basic region* W , and:
 - (a) W is already locked by this programmer (granted and recorded previously) as a *working region*; or
 - (b) W is not locked by any other collaborating programmer, and none of W 's *depended regions* is locked as a *working region* by any other collaborating programmer.

In Definition 5.1, Condition 2(b) ensures that the shared-locks are placed on depended regions only. For example in Figure 5.6, if another programmer P_3 attempts to edit region E which depends on region D , the editing permission will not be granted because the depended region D is already locked as a working region by P_2 .

As the shared-locking among common depended regions has already been allowed and justified under circumstances in which no concurrency is involved and the DG remains static, the D-D Overlapping can actually be allowed under all circumstances, including the cases where concurrency is involved or there exist dynamic DG editing operations. In the following two sub-sections for supporting shared-locking for concurrent and dynamic DG editing operations, the D-D Overlapping is always allowed and hence will not be explicitly mentioned.

5.3.2 Shared-Locking for Concurrent Editing Operations with Overlapping Locking Scopes

We extend the basic DAL scheme to allow *shared-locking* on *overlapping locking scopes* caused by *concurrent* editing operation. Such lock-sharing may happen under two circumstances:

1. When multiple programmers concurrently edit *the same free region*, they will be granted shared-locks on the overlapping working region, denoted as *W-W Overlapping*. Upon the granting of the shared-locks, the programmers involved will be notified of who is/are sharing. For example, Figure 5.7 illustrates the lock-sharing on the overlapping working region A.
2. When multiple programmers concurrently edit *free regions with dependency relationships*, they will be granted shared-locks on the overlapping working region and depended region, denoted as *W-D Overlapping*. Upon the granting of shared-locks, the programmers involved will be notified of who is/are sharing. For example, Figure 5.8 illustrates the lock-sharing on the working region and depended region B.

Figure 5.7: Shared-locking for concurrent editing operations (W-W Overlapping)

Figure 5.8: Shared-locking for concurrent editing operations (W-D Overlapping)

The rationales for allowing shared-locking by W-W Overlapping and W-D Overlapping for concurrent editing operations are presented as follows:

1. The shared-locking scheme is able to achieve consistent locking states at all collaborating sites in the presence of concurrent editing operations with overlapping locking scopes without sacrificing local responsiveness.
2. The shared-locking is combined with a notification mechanism to inform the involved programmers of the lock-sharing situations, and provides flexible options to those programmers: they may choose to (1) continue the concurrent access to the same or interdependent regions by holding shared-locks if they are able to take care of the issues related to the shared-locking; or (2) negotiate and collectively decide who should be given the exclusive editing permission at the moment.

5.3.3 Shared-Locking for Dynamic DG Editing Operations

We extend the basic DAL scheme to allow *shared-locking on overlapping locking scopes* created or expanded by *dynamic* DG editing operations. In case of lock-sharing, programmers will be notified of who is/are sharing. The two motivating rationales in the shared-locking for concurrent editing operations are also applicable to shared-locking for dynamic DG editing operations. In addition, the shared-locking for dynamic DG editing operations has another useful feature in interactive programming environments: a programmer, once obtained the initial permission granted by the locking system, is able to work continuously in the presence of dynamic DG changes, without any disruption.

For example in Figure 5.9 which re-illustrates the case in Figure 5.4, P_1 initially got the permission to work in source code region A with locks granted on working region A and

depended region B , and P_2 initially got the permission to work in source code region C with locks granted on working region C and depended region D .

Figure 5.9: Shared-locking for dynamic DG editing operations (without concurrency involved)

Under the shared-locking scheme, P_1 is able to work continuously in source code region A , even after P_1 's locking scope has been expanded to cover the new depended region C (which is already locked for P_2 as a working region). Similarly, although P_1 's depended regions have covered the source code region C which is currently a working region locked for P_2 , P_2 can still work continuously in source code region C .

Provided that a programmer initially obtained the editing permission to work in a source code region and stayed working in that region without switching to other source code regions or open areas, the editing permission will never be withdrawn by the DAL sys-

tem. With the additional notification mechanism for informing all involved programmers of the lock-sharing situation, collaborating programmers can negotiate and decide what coordination is needed or simply continue the programming work with the awareness of potential semantic consistency issues under the shared-locking situation.

Similarly, in Figure 5.10 which re-illustrates the case in Figure 5.5, both P_1 and P_2 can continue working in the newly created source code region A by holding shared-locks on the common working region. This case is more complex than the case in Figure 5.9 as the shared-locking is caused by a combination of both concurrent editing operations and dynamic DG editing operations. However, from the programmers' perspective, it is not necessary to differentiate shared-locking circumstances caused by different reasons. The only thing that is meaningful to their collaborative programming work is to make them aware of being involved in lock-sharing situations by means of the notification mechanism, so that they can take actions accordingly.

Figure 5.10: Shared-locking for dynamic DG editing operations (in combination with concurrency)

5.3.4 Common Design Rationales for Three Types of Shared-Locking

One uniform design rationale for all three types of shared-locking is to preserve the basic DAL semantics under all circumstances: when a programmer is permitted to work in a source code region, the system must preserve locks on this working region and all its depended regions (if any) to this programmer. In other words, the locking scope of any programmer must always cover the latest working region and depended regions. This property is essential for the DAL system to support all types of shared-locking by applying the same set of mechanisms, which are effective and efficient, and also verifiable with respect to the same set of locking state correctness criteria (to be discussed in Section 5.4).

In addition, another common design rationale for shared-locking by both concurrent and dynamic DG editing operations is to preserve the continuous work (i.e., without withdrawing existing locks) for any programmer under all circumstances provided that the programmer has initially got the editing permission in the working region and keeps working in the region without switching to other locations. For shared-locking by concurrent editing operations, once a programmer has initially got the editing permission granted by the local permission check procedure, the granted locks will not be withdrawn when another lock is concurrently placed on the same working region or its depended region by a remote programmer. For shared-locking by dynamic DG editing operations, once a programmer has initially got the editing permission for a working region, the granted locks will not be withdrawn when any changed dependency relationship (i.e., by dynamic DG editing operations) expands the locking scope to cover certain regions that are locked by others.

It is worth highlighting that the shared-locking scheme designed for the extended DAL scheme is different from those in conventional database and distributed systems where programmed processes holding shared-locks are permitted to *read* but not to *write* the locked data/object. In real-time collaborative programming environments with the DAL mechanisms, the *shared-read-permission* to a source code region is granted to all collaborating programmers by default (i.e., all contents in the same source code file can be concurrently viewed by all programmers within the session). The DAL mechanism only controls the *write-permission* to various source code regions. A shared-lock in the DAL mechanism corresponds to the *shared-write-permission* to a source code region, and programmers holding shared-locks are allowed to *write* the shared-locked regions. In the DAL mechanism, whether a lock is *exclusive* or *shared* is not determined by the lock itself but by the *locking context* of the region: a lock is *exclusive* if it is the unique lock granted on the region, or *shared* if there are multiple locks granted on the same region.

Shared-locking with notification is most suitable for multi-user real-time collaborative systems. In contrast to conventional database and distributed systems where locks are used to synchronize programmed processes, locks in real-time collaborative programming are used to coordinate actions among intelligent human users, who are capable of avoiding conflicts and adapting themselves to resolve potential conflicts.

Moreover, this shared-locking scheme can be efficiently implemented without sacrificing local responsiveness: for a local user's action, the DAL system can quickly decide whether the user is permitted to edit the targeted region and grant corresponding locks solely based on the information recorded in the local locking state. Once the editing permission (by DAL locks) is granted locally, the same permission can be guaranteed at all remote sites due to the adoption of the shared-locking for concurrent editing opera-

tions, and the DAL system at a remote site can easily accommodate shared-locking in the local locking state by simply determining and granting locks based on the arriving editing operation.

5.4 The Extended DAL Scheme

In this section, we present the extended DAL scheme, which has been designed with the shared-locking scheme as the cornerstone based on the revised permission check conditions. Firstly, a set of extended DAL locking state data structures for accommodating shared-locking under the extended DAL scheme are presented in Section 5.4.1. To guide the design of locking state update mechanisms under the extended DAL scheme, a set of locking state correctness criteria are proposed in Section 5.4.2. For technically realizing the extended DAL scheme, a *Contextualization and Full Derivation* (CFD) locking state update scheme is proposed in Section 5.4.3. Following that, key algorithms and procedures for the extended DAL scheme are presented in Section 5.4.4, and the correctness verification of the CFD locking state update scheme with respect to the set of correctness criteria is presented in Section 5.4.5.

5.4.1 Extended Locking State Data Structures for Accommodating Shared-Locking

For supporting the extended DAL scheme, a three-level hierarchical data structure is designed for DAL locking state maintenance as illustrated in Figure 5.11, which has extended the basic DAL locking state data structures in Section 4.6.4. Firstly, the source code is associated with a *DAL Table* at the root level; secondly, the *DAL Table* consists of a list of *DAL Regions* at the middle level; thirdly, each *DAL Region* contains a set of

DAL Locks held by different programmers at the leaf level. The data structure for each *DAL Region* is able to accommodate multiple *DAL Locks*, which has been designed to allow multiple locks held by different collaborating programmers to be placed on the same source code region for supporting the shared-locking scheme.

Figure 5.11: Three-level hierarchical data structures for accommodating shared-locking under the extended DAL scheme

In the locking state data structures, the *DAL Lock* represents a lock granted to a programmer on a source code region, which is expressed as $\langle Owner\ ID, Region\ Type \rangle$, where the *Region Type* indicates whether the locked region is a working region (denoted as *W*) or a depended region (denoted as *D*) with respect to the owner. The *DAL Region* represents a source code region (e.g., method/field) that is currently locked by programmers, which is expressed as $\langle Region\ Reference, Lock\ List \rangle$, where the *Region Reference* relates the *DAL Region* to the corresponding element in the source code contents, and the *Lock List* stores a list of *DAL Locks* placed on this source code region.

Figure 5.12 illustrates an instance of the DAL locking state data structures at one moment when two programmers P_1 and P_2 are editing basic regions *A* and *E* respectively in a real-time collaborative programming session. As indicated in the *DAL Table*: (1) locks on working region *A* and depended regions $\{B, C, F\}$ are granted to P_1 ; and (2) locks on

working region E and depended regions $\{C, F\}$ are granted to P_2 . As presented in the DG, five source code regions $\{A, B, C, E, F\}$ are currently locked by programmers (and thus involved in the DAL mechanism), so there are five corresponding *DAL Regions* within the *DAL Table*.

Figure 5.12: An instance of the DAL locking state data structures

The three utility functions related to DAL locking states, including the *CheckOwnership*(R, U, S, T) function presented in Table 4.1 of Section 4.7.1 and the *GrantLocks*(W, DRS, U) and *ReleaseLocks*(U) functions presented in Table 4.2 of Section 4.7.2, need to be revised accordingly based on the extended DAL locking state data structures, which are presented as follows.

The *CheckOwnership* function firstly searches for the *DAL Region* with respect to the specified source code region R by comparing R with the *Region Reference* of each *DAL Region* within the *DAL Table*. Once the particular *DAL Region* is found in the table, the *DAL Locks* within the *Lock List* of this *DAL Region* are further visited and examined to judge the ownership with respect to the specified conditions. In the worst case, all basic regions are involved in the *DAL Table*, and the locks may be shared by all collaborating users in the real-time collaboration session. Therefore, the complexity of *CheckOwnership* is $O(N + U)$ where N is the amount of basic regions and U is the number of collaborating users. The runtime overheads of *CheckOwnership* mainly consist of the time spent on searching for the *DAL Region* and the time spent on visiting the *DAL Locks* within the *Lock List* if the particular *DAL Region* is found in the table.

The *GrantLocks* function inserts new entries (i.e., *DAL Locks*) into the *DAL Table* in accordance with the specified working region W and depended regions in the *DRS*. For each basic region among them, the function firstly searches for the *DAL Region* that relates to the basic region. If the particular *DAL Region* is found, a *DAL Lock* will be inserted within its *Lock List*; otherwise, if no corresponding *DAL Region* is found, a new *DAL Region* will be inserted into the *DAL Table*, and then a *DAL Lock* will be inserted within the *Lock List* of this new *DAL Region*. The complexity of *GrantLocks* is $O((1 + |DRS|) * K)$ where K is the amount of *DAL Regions* within the *DAL Table*. In the worst case, $1 + |DRS| = N$ and $K = N$ where N is the amount of basic regions in the source code. Therefore, the complexity of *GrantLocks* can be expressed as $O(N^2)$. The runtime overheads of the function completely depend on the size of the *DRS*, as the searching and insertion operations are executed for exactly $(1 + |DRS|)$ times.

The *ReleaseLocks* function visits all *DAL Locks* within the *Lock Lists* of all *DAL Re-*

gions. In the worst case, the amount of *DAL Regions* equals to the amount of basic regions (denoted as N), and the amount of *DAL Locks* within the *Lock List* of each *DAL Region* equals to the number of collaborating users (denoted as U). The complexity of *ReleaseLocks* is therefore $O(NU)$, and the runtime overheads of the function completely depend on the total amount of *DAL Locks* within the *DAL Table*.

5.4.2 Correctness Criteria for Locking State Maintenance

Under the extended DAL scheme, locking state maintenance is the key to realizing unconstrained real-time collaborative programming with the shared-locking. To guide and verify the design of the locking state update scheme and techniques, we propose a set of DAL locking state correctness criteria as follows.

Definition 5.2: DAL Locking State Correctness Criteria

Under the extended DAL scheme, locking states in a real-time collaboration session are correct if the following conditions are met:

1. *Convergence*: after executing the same group of editing operations, the locking states are identical at all sites within the session.
2. *Basic DAL Correctness*: a source code region is locked by a programmer *if and only if* this region is: (i) a working region of this programmer; or (ii) a depended region with respect to the working region of this programmer.
3. *DAL Shared-Locking Constraints*: a source code region can be locked by multiple programmers *only if*: (i) it is shared as a depended region for all involved programmers; or (ii) it is shared as a working region for at least one programmer due to *concurrent* editing operations or *dynamic* DG editing operations.

The *Convergence* criterion ensures that the same locking states are produced at all sites

after executing the same group of editing operations, which is necessary for achieving consistent permission check at all sites. Convergence is a necessary and generic requirement, but insufficient for ensuring correctness under the DAL scheme. Two more DAL-specific criteria must be met for ensuring the correctness of locking states.

The *Basic DAL Correctness* criterion ensures compliance with the basic DAL semantics in determining the locking scope for a programmer: during a real-time collaboration session, the locking scope for a programmer must always exactly cover the latest working region (if not in an open area) and depended regions (if any). The locking scope for a programmer should never cover any source code region that is currently not a working or depended region of the programmer. Whenever the locking scope needs to be updated for a programmer, the working region and all its depended regions should be locked or unlocked together under all circumstances.

In compliance with the shared-locking scheme, the *DAL Shared-Locking Constraints* criterion ensures that the lock-sharing may only occur under specific conditions. As constrained by the shared-locking scheme in Section 5.3, the extended DAL scheme strictly enforces the permission check with respect to each local editing operation, but allows different types of shared-locking under different conditions as summarized in Table 5.1: (1) *D-D Overlapping* is always allowed under any condition; and (2) *W-W Overlapping* or *W-D Overlapping* is allowed by either *concurrent* editing operations or *dynamic* DG editing operations, or a *combination* of both. In other words, a depended region can be locked by multiple programmers under all circumstances, whereas a working region can be locked by multiple programmers only if the lock-sharing (overlapping) is caused by *concurrent* or *dynamic* DG editing operations.

Table 5.1: Different types of shared-locking allowed under different conditions

	<i>Concurrent Editing Operations</i>	<i>Sequential Editing Operations</i>
<i>Dynamic DG Editing Operations</i>	<i>W-W Overlapping</i> <i>W-D Overlapping</i> <i>D-D Overlapping</i>	<i>W-W Overlapping</i> <i>W-D Overlapping</i> <i>D-D Overlapping</i>
<i>Static DG Editing Operations</i>	<i>W-W Overlapping</i> <i>W-D Overlapping</i> <i>D-D Overlapping</i>	<i>D-D Overlapping</i>

5.4.3 Locking State Update under the Extended DAL Scheme

Under the extended DAL scheme, locking state update needs to be conducted after the execution of each editing operation (either local or remote), because editing operations may change the DG structure and affect the locking scopes of collaborating programmers. The locking state update procedure analyzes the working region, derives the depended regions, and updates the locks granted to the programmers. For the locking state update mechanism itself, there is no need to take care of whether a granted lock is exclusive or shared, as such property of the lock solely depends on the current locking context on that region (i.e., locks placed on the region), which is just the outcome produced by the locking state update.

There are two approaches to updating the locking state: (1) *operation-based update* and (2) *state-based update*. Under the *operation-based update* approach, the locking state is updated incrementally by extracting fine-grained DG changes embedded in each editing operation. This approach is able to support fine-grained locking state update, but has a

major problem in practice: as discussed in Section 5.2.3, a single editing operation may trigger complex and compound DG changes, which essentially requires the capability of parsing the whole source code.

Under the *state-based update* approach, the locking state is updated based on: (1) the latest document state of the source code (as well as the implicitly embedded DG); and (2) the latest *editing positions* of programmers (i.e., working regions or open areas in which programmers are working). We adopt the *state-based update* approach for the following reasons. Firstly, the *dependency derivation technique* (as discussed in Section 4.6.3) for deriving source code regions and dependency relationships is efficient enough for invocation after every editing operation performed. Secondly, since the same derivation technique is applied at multiple sites, the consistent locking state update can be naturally guaranteed. Thirdly, the design of the *DAL Table* data structures (as presented in Section 5.4.1) makes it efficient to record the updated locks by simply replacing the originally recorded locks with the newly derived locks.

One key element of the state-based locking state update is the *editing position*, which is used to determine the latest location (i.e., working region or open area) where a programmer is working. Let $EP(i)$ be the latest editing position of Site i based on the current source code document state, which ranges from 0 to $(L - 1)$ where L is the length of the source code contents. Under the extended DAL scheme, each collaborating site in a real-time file session maintains an *editing position list* $\{EP(i)\}$ (where $i = 1$ to N) for all N collaborating sites in the session.

For each editing operation performed on the source code, a *contextualization* procedure is applied to update the editing positions of all programmers in the session based on the

new document context (document state). Consider the example in Figure 5.13 where there are three active collaborating sites in a real-time file session, and the initial editing positions of the programmers are indicated in the upper part. After an editing operation $Insert[2, "abc"]$ (to insert the string "abc" at the position 2) has been performed on the source code, the document state evolves into what is presented in the lower part. As illustrated, the editing operation issued by P_1 not only changes the editing position of P_1 but also changes the editing positions of P_2 and P_3 . If those editing positions are not adjusted or contextualized based on the new document state, derivations of basic regions and dependency relationships based on the old editing positions would be problematic. For example, if P_2 's editing position is not contextualized, the derived working location for P_2 would fall into an open area, which will lead to incorrect locking state update. Therefore, for supporting locking state update under the extended DAL scheme, such contextualization procedure will be applied after each editing operation has been performed, which updates the editing positions of all active collaborating sites of the session to produce contextualized editing positions for all collaborating programmers.

Figure 5.13: Contextualization of editing positions

In the absence of concurrent or dynamic DG editing operation, locking state update under the basic DAL scheme (as designed in Section 4.7.2) can be achieved by a *partial derivation* approach: for each editing operation performed, only the *operation-actor*'s² locking state is updated. To elaborate, for each *local* editing operation performed, the system only updates the locking state of the *local* site; for each *remote* editing operation executed, the system only updates the locking state of the corresponding *remote* site that generates the operation.

It has been observed that the *partial derivation* approach is not able to support correct locking state update in the presence of concurrent and dynamic DG editing operations. Consider a simple example illustrated in Figure 5.14 where two programmers P_1 and P_2 are concurrently working at the same location (initially an open area) of the source code. The proposed method *method()* is initially invalid due to a missing right brace, so both of them are working in an open area without any lock granted at the beginning.

At one moment, they concurrently edit the source code as follows: (1) P_1 attempts to delete the incomplete piece of source code and create another method, and starts by deleting the left brace; (2) P_2 attempts to complete the method by inserting the right brace, which transforms the piece of source code into a valid source code region, and consequently, a lock on the newly created region *A* is immediately granted to P_2 by the locking state update procedure.

² The *operation-actor* of an editing operation refers to the site which generates the operation, whereas a *non-operation-actor* of an editing operation refers to a remote site that replays the operation.

Figure 5.14: Locking state update by partial derivation

Under the *partial derivation* approach, no locking state update is performed for the local site when a remote editing operation has been executed. The final locking state at P_1 's site is correct by accident, whereas the final locking state at P_2 's site is incorrect because a lock is still granted to P_2 on a non-existing source code region; and the final locking states at the two sites are inconsistent. The essence of the problem is that a dynamic DG editing operation may affect both the *operation-actor* and *non-operation-actors*' locking states, but the partial derivation approach does not update non-operation-actors' locking states.

To solve this problem, a *full derivation* approach is proposed: for each editing operation performed (either local or remote), locking states of all active sites of the session (including the *operation-actor* and all *non-operation-actors*) are *fully* derived and updated. As illustrated in Figure 5.15, the *full derivation* approach is able to produce correct and consistent locking states at all collaborating sites in the presence of concurrent and dynamic DG editing operations.

Figure 5.15: Locking state update by full derivation

To summarize, locking state update under the extended DAL scheme involves two major steps: (1) to contextualize and update the editing positions of all collaborating sites; and (2) to derive and update the locking states of all collaborating sites based on the current document state and contextualized editing positions. The locking state update scheme is therefore named as a *Contextualization and Full Derivation (CFD) scheme*, which is formally defined as follows.

Definition 5.3: Contextualization and Full Derivation (CFD) Scheme

At each collaborating site of a real-time file session, for every local or remote editing operation performed on the source code, locking states for all collaborating sites are derived and updated as follows:

For each collaborating site i of the session (where $i = 1$ to N):

Step 1. All original locks (if any) granted to Site i are released.

Step 2. $EP(i)$ is derived by the contextualization procedure.

Step 3. If $EP(i)$ falls into a basic region W , locks on working region W and its depended regions are granted to Site i ; otherwise, no lock is

granted to Site i for editing in an open area.

An important property of the CFD scheme is that the task of updating each site's locking state is *order-independent*, because: (1) the shared source code contents are identical over all collaborating sites; and (2) the editing position of each collaborating site is independent of others' editing positions.

Figure 5.16 presents a comprehensive example that illustrates how the CFD locking state update works step by step.

Figure 5.16: A comprehensive example of CFD locking state update

There are three programmers (P_1 , P_2 , and P_3) working concurrently at different locations of the same source code file, and three editing operations are issued at different moments: O_1 is issued by P_1 which creates a dependency relationship $A \rightarrow C$; O_2 is issued by P_2 which destroys the source code region C ; and O_3 is issued by P_3 which simultaneously creates a source code region E and a dependency relationship $E \rightarrow A$. At each collaborating site of the session, the CFD locking state update is executed with respect to every local or remote editing operation performed, and finally the consistent and correct locking states are produced over all collaborating sites.

5.4.4 Key Procedures for the Extended DAL Scheme

Based on the revised permission check conditions in Definition 5.1 and the extended locking state data structures in Section 5.4.1, the *permission check procedure* can be devised for the extended DAL scheme accordingly. As the essential difference between the permission check conditions under the basic DAL scheme and those under the extended DAL scheme lies on the shared-locking on common depended regions, the permission check procedure for the extended DAL scheme can be devised by revising the permission check procedure for the basic DAL scheme in Section 4.7.1, which is implemented by Algorithm 5.1 below.

Algorithm 5.1: *PermissionCheck*(O, U)

```

W := GetWorkingRegion(O);
if W = NULL
    return PERMIT_OA;
if CheckOwnership(W, U, SELF, WORKING) = TRUE
    return PERMIT_WR;
if CheckOwnership(W, U, OTHERS, ALL) = TRUE

```

```

        return REJECT;
DRS := GetDepRegionSet(W);
for each Basic_Region D in DRS {
    if CheckOwnership(D, U, OTHERS, WORKING) = TRUE
        return REJECT;
}
return PERMIT_FR;

```

Similar to the complexity analysis for the *PermissionCheck* procedure under the basic DAL scheme, the complexity of the *PermissionCheck* procedure here can be derived based on the complexities of the *GetWorkingRegion*, *CheckOwnership* and *GetDepRegionSet* utility functions.

Suppose that N is the amount of basic regions, M is the total amount of field references and method invocations, and U is the number of collaborating users. As discussed in Section 5.4.1, the complexity of *CheckOwnership* is $O(N + U)$ under the extended DAL scheme, whereas the complexities of *GetWorkingRegion* and *GetDepRegionSet* remain the same. The overall complexity of *PermissionCheck* is $O(N + 2(N + U) + NM + N * (N + U))$, which can be simplified as $O(NM + N^2 + NU)$.

Based on the extended locking state data structures, the shared-locking scheme, and the CFD locking state update scheme, the *locking state update procedure* for the extended DAL scheme can be devised accordingly. Algorithm 5.2 implements the CFD locking state update procedure, which will be executed at all sites with respect to each local or remote editing operation performed on the source code. As implemented, for each collaborating site, its locking state is derived and updated solely based on the latest source code document state and the contextualized editing position.

Algorithm 5.2: *CFD_LSUpdate()*

```
for each site U in the real-time file session {
    ReleaseLocks(U);
    W := GetWorkingRegion(EP(U));
    if W != NULL {
        DRS := GetDepRegionSet(W);
        GrantLocks(W, DRS, U);
    }
}
```

Based on the algorithm presented, the complexity of *CFD_LSUpdate* is $O((NU + N + NM + N^2) * U)$, which can further be simplified as $O(NU^2 + NMU + N^2U)$ where N is the amount of basic regions, M is the total amount of field references and method invocations, and U is the number of collaborating users.

Algorithm 5.3 implements the integrated local operation handler *LOH*(O, U), which processes an editing operation O issued by the local user U . The handler first invokes the permission check procedure. If the editing permission is denied, O is prohibited from execution, and U is notified of the failure; otherwise, O is executed on the local source code, and the CFD locking state update procedure is then executed to update the locking states accordingly. Finally, the editing operation is propagated to remote sites.

Algorithm 5.3: *LOH*(O, U)

```
result := PermissionCheck(O, U);
if result = REJECT {
    Reject Execution of O and Notify User U;
    return;
}
```



```

}
Execute (O) ;
CFD_LSUpdate () ;
Propagate (O) ;

```

Algorithm 5.4 implements the integrated remote operation handler $ROH(O, U)$ for processing an editing operation O issued by a remote user U . Firstly, O is executed without permission check because it has been permitted at its local site and its permission at a remote site is guaranteed under the shared-locking scheme. The CFD locking state update procedure is then invoked after the execution of the remote editing operation.

Algorithm 5.4: $ROH(O, U)$

```

Execute (O) ;
CFD_LSUpdate () ;

```

In addition, all locking release mechanisms for the basic DAL scheme designed in Section 4.7.4 can be reused with the same semantics under the extended DAL scheme:

1. Whenever a new set of locks are derived and granted to a programmer by the locking state update procedure, the original set of locks held by this programmer will be automatically released (before granting the new locks);
2. If a programmer has not generated any operation in the programming environment for a pre-set duration of time, all locks held by this programmer will be automatically released (integrated with notification features such as the countdown timer displayed in the user interface);

3. A programmer may also manually issue an operation provided in the user interface to release locks that are currently held by this programmer.

Major technical solutions for supporting these three mechanisms under the extended DAL scheme are similar to those under the basic DAL scheme. The first mechanism is achieved by nature in the CFD locking state update procedure. In terms of the second and third mechanisms, for either the time-out-based automatic locking release or the manual locking release, a particular communication message for indicating the locking release operation will be propagated to remote sites. At a remote site, whenever such a message arrives and instructs the system to release locks for the specified site under the extended DAL scheme, locks granted to this site will be removed from the *DAL Table*, and the *editing position* of this site will be set to a special value (e.g., -1) to indicate that this site is currently not editing at any location. By this measure, the CFD locking state update procedure can check whether the editing position of a site is in such a special value and determine whether it is needed to derive and grant locks for this site accordingly.

5.4.5 Correctness Verification of the CFD Locking State Update Scheme

In this section, we verify the correctness of the CFD locking state update scheme with respect to the set of locking state correctness criteria specified in Definition 5.2. For the convenience of verification, we first introduce a lemma and present its proof as follows.

Lemma 1: The *DAL Shared-Locking Constraints* criterion in Definition 5.2 is satisfied if no working region of one programmer is locked by another programmer due to *sequential* or *static* DG editing operations.

Proof: In the *DAL Shared-Locking Constraints*, there is no restriction imposed on the shared-locking among depended regions as it is allowed under any circumstance; and the shared-locking on a working region of one programmer is always allowed due to either *concurrent* or *dynamic* DG editing operation. Therefore, the *DAL Shared-Locking Constraints* are equivalent to a single constraint, i.e., no working region of one programmer is locked by another programmer due to *sequential* or *static* DG editing operations, from which the lemma follows.

The following theorem establishes the correctness of the CFD locking state update scheme.

Theorem 1: The locking states produced by the Contextualization and Full Derivation (CFD) locking state update scheme satisfy all locking state correctness criteria specified in Definition 5.2.

Proof: We prove the satisfaction of the three correctness criteria one by one as follows:

1. *Convergence:* After the same group of editing operations have been executed in their causal orders at all sites of a session, the source code file contents, the implicitly embedded DG, and the contextualized editing positions of all collaborating sites must be identical over all sites as guaranteed by the OT technique for syntactic consistency maintenance. In addition, as the same techniques and procedures are applied at all sites to derive and update the locking states based on the identical source code file contents and consistent contextualized editing positions, the locking states produced must be the same at all sites.
2. *Basic DAL Correctness:* Step 3 of the CFD locking state update scheme (in Defini-

tion 5.3) ensures the *Basic DAL Correctness*.

3. *DAL Shared-Locking Constraints*: Under the DAL scheme, the generation of each editing operation at its local site is constrained by the DAL *permission check*, and the execution of the operation at remote sites is constrained by *causal-ordering* among editing operations. Constrained by the DAL permission check, when a programmer starts to edit a source code region which is currently not locked by this programmer as a working region, the programmer can be allowed to perform local editing operations in the region (together with locks granted) only if this region is not locked by another programmer and no depended region is locked by another programmer as a working region, which implies that no working region of a programmer is locked by another programmer in the local locking state before executing the operation. If the editing operation is static, the locking state will remain unchanged after executing the operation because a static DG editing operation never changes the DG structure or the locking state. If there is no concurrency involved (i.e., sequential operations only), with the enforcement of the causally-ordered execution guaranteed by the underlying mechanism, all sequential editing operations are executed in the same order at all collaborating sites. After executing a locally-permitted editing operation at a remote site, the locking state at the remote site must be the same as the local locking state because the convergence of locking states has been guaranteed. In conclusion, with the DAL permission check and the causally-ordered execution, no working region of a programmer can be locked by another programmer at local or remote sites by sequential or static DG editing operations, which implies the satisfaction of the *DAL Shared-Locking Constraints* by *Lemma 1*.

5.5 Comparison with Related Work

Locking is the essential element of the DAL approach and scheme, which plays an important role in supporting semantic conflict prevention. In this section, we provide a set of comparisons between the DAL scheme in this research and other locking schemes in related work.

Firstly, we compare the DAL scheme with the locking schemes in traditional database systems and distributed computing environments. One common point is that the locking is applied to achieve mutual exclusion: in the DAL scheme, mutual exclusion helps to prevent semantic conflicts by prohibiting concurrent editing on the same or interdependent source code regions; in database and distributed systems, mutual exclusion ensures data integrity by prohibiting concurrent updates on shared objects [15]. Compared to those traditional locking schemes, the DAL scheme can be distinguished in the following aspects:

1. Locks granted under the DAL scheme are used to coordinate the collaborative work among intelligent human users (programmers), whereas locks granted in database and distributed systems are used to synchronize programmed processes.
2. Locks under the DAL scheme are requested in a non-blocking manner and thus the deadlock is impossible, whereas a programmed process issuing a locking request will be blocked until the result is obtained and thus there exists the risk of deadlock under these traditional locking schemes.
3. The shared-read-permission to all source code contents is granted to all programmers by default and the DAL scheme only controls the write-permission to selected

source code regions, whereas the traditional locking schemes in database and distributed systems may control both types of permissions by read-lock and write-lock.

Secondly, we compare the DAL scheme with the locking schemes for supporting syntactic consistency maintenance in collaborative editing systems. In some early collaborative editing systems, locking was used for achieving consistency maintenance of the shared documents [10][46][54][66][70][71][72]. In these collaborative editing systems, any document object must be locked before it is edited, so each object can be edited by at most one user at any time, while concurrent editing is allowed on different objects. All locks are compulsory and exclusive in these early systems as they were mainly used for supporting syntactic consistency maintenance. However, further studies had discovered that locking is not capable of solving any syntactic consistency problem, unless the granularity of the locking is the entire document (i.e., thus preventing concurrent work) [92][95][96]. In this research, locking is used for supporting semantic conflict prevention only, whereas the syntactic consistency maintenance is supported by the OT technique combined with other distributed computing techniques.

Thirdly, we compare the DAL scheme with the locking schemes for supporting semantic consistency maintenance in collaborative editing systems. Prior work has contributed an optional and responsive fine-grained locking scheme for semantic consistency maintenance in real-time collaborative editing systems [92][96]. Locking was made optional in order to avoid the locking overhead in the most common cases of collaborative editing, and the OT technique has been extended to support both editing and locking operations. The DAL scheme presented in this research is a significant extension to the prior work in the following major aspects:

1. Locks can be automatically derived, granted and released under the DAL scheme, whereas locks should be manually manipulated in prior work.
2. Locks are granted as a group of locks on interdependent regions where the dependency relationships are defined on complex source code structures under the DAL scheme, whereas locks are granted as individual locks on single text regions defined on plain text documents with the linear addressing scheme in prior work.
3. Locks can be shared among collaborating programmers in different forms based on different region types and conditions under the DAL scheme, whereas locks can only be shared due to concurrent and overlapping locking operations (or cannot be shared) in prior work.

5.6 Summary

In this chapter, we contributed an extended DAL scheme, which extends the basic DAL scheme to support semantic conflict prevention in unconstrained real-time collaborative programming where programmers are allowed to concurrently edit source code regions with overlapping locking scopes and dynamically change the DG structure of the source code.

To address the three major restrictions of the basic DAL scheme in supporting unconstrained collaboration, we proposed a *shared-locking* scheme. The scheme allows three different types of shared-locking under well-defined circumstances, in which the effectiveness of semantic conflict prevention is preserved. Firstly, shared-locking is allowed on common depended regions under any circumstance; secondly, shared-locking is allowed on overlapping locking scopes for concurrent editing operations; and thirdly,

shared-locking is allowed on overlapping locking scopes for dynamic DG editing operations. The shared-locking scheme is capable of preserving basic DAL semantics on locking scope determination, preserving continuous work for any programmer who is holding the editing permission on the working region, and supporting high local responsiveness with minimum overheads on system resource and communication bandwidth. A notification mechanism is integrated with the shared-locking mechanism to facilitate awareness and negotiation among programmers in case of lock-sharing.

With the shared-locking scheme as the cornerstone, we presented the extended DAL scheme with supporting techniques. Firstly, we extended the locking state data structures for accommodating shared-locking on source code regions. Secondly, we formulated three correctness criteria for guiding the design of the locking state update scheme, which is the key to supporting the extended DAL scheme. Thirdly, in recognizing the impacts of editing operations on both operation-actor and non-operation-actors' locking states, we devised a contextualization and full derivation (CFD) locking state update scheme, in which the locking states of all collaborating sites in a session will be fully derived and updated based on the latest document state and the contextualized editing positions, with respect to each local or remote editing operation performed. Key procedures for the extended DAL scheme have been devised accordingly, including the revised permission check procedure, the CFD locking state update procedure, and the integrated operation handlers for processing local and remote editing operations. In addition, we presented the correctness verification of the CFD locking state update scheme with respect to the locking state correctness criteria.

In the next chapter, we will discuss major technical issues involved in building an ATCoPE system, and present the design and implementation of a research prototype

named *ATCoEclipse*, which has realized all major technical solutions devised in this research and serves as a proof-of-concept of ATCoPE.

Chapter 6.

The ATCoEclipse Prototype System: A Proof-of-Concept of ATCoPE

6.1 Introduction

In previous chapters, we contributed the design of the ATCoPE system architecture and a package of major techniques for supporting ATCoPE, which collectively convert the abstract notion of any-time collaborative programming paradigm into concrete system functionalities and features. In this chapter, we present the design and implementation of a research prototype system named *ATCoEclipse (Any-Time Collaborative Programming with Eclipse)*, which serves as a proof-of-concept of ATCoPE. All major technical approaches and solutions devised in this research have been implemented in the ATCoEclipse system, which provides positive confirmation of their technical feasibility and preliminary performance feedback.

In addition to discussing the techniques for implementing the ATCoEclipse prototype, we also present the design of major user interfaces of the ATCoEclipse Client, and contribute a novel DAL-related collaboration awareness feature to support locking awareness in the presence of the DAL scheme, which complements the locking mechanism in collectively supporting semantic conflict prevention.

The rest of this chapter is organized as follows. In Section 6.2, we discuss the general approach for implementing the ATCoEclipse system. In Section 6.3, we present the design and implementation of the ATCoEclipse system. In Section 6.4, we present the design of major user interfaces of the ATCoEclipse Client as well as the DAL-related col-

laboration workspace and locking awareness feature. In Section 6.5, we present preliminary performance feedback and analysis of the ATCoEclipse system, including a set of micro benchmark experiments. Finally, in Section 6.6, we summarize this chapter.

6.2 The Transparent Adaptation Approach for ATCoEclipse System Implementation

For implementing the ATCoEclipse system, we have adopted the *Transparent Adaptation* (TA) approach, which is a generic approach for converting existing single-user applications into multi-user collaborative applications without modifying the source code of the original applications [98]. TA-based multi-user collaborative applications are capable of supporting both rich conventional functionalities available from existing single-user applications and novel collaboration functionalities derived from emerging research. To the end-users, the transparency means that they are able to use the same system and environment with the same knowledge, experience and working process as before while enjoying extra collaboration capabilities. To system builders and researchers, the transparency means that they can focus on implementing innovative techniques and efficiently build multi-user collaborative applications without reinventing existing techniques for conventional single-user functionalities or existing collaboration capabilities.

In the past studies, the TA approach has been successfully applied to transparently building multi-user real-time collaborative two-dimensional (2D) office applications (such as *CoWord* and *CoPowerPoint* [98][111]) and three-dimensional (3D) digital media design tools (such as *CoMaya* [4]) from existing single-user 2D/3D applications. In this research, the TA approach is applied for building any-time collaborative programming systems by integrating existing single-user and non-real-time collaborative pro-

programming systems with novel real-time collaborative programming techniques which support both syntactic and semantic consistency maintenance features. With the TA approach, those conventional and sophisticated single-user programming functionalities can be supported in the ATCoEclipse system without reinvention, and the end-users can use the same functionalities and user interfaces for single-user programming as before while enjoying advanced real-time collaboration functionalities with syntactic and semantic consistency maintenance features.

For supporting single-user programming functionalities, we have chosen the Eclipse IDE for the following reasons. Firstly, Eclipse is an open platform and framework, which is structured by various sub-systems as plug-ins with a runtime core, thus allowing developers to add new functionalities and features based on the platform. Secondly, Eclipse contains various plug-in sets with rich extensibilities in the form of programming interfaces for other plug-ins to utilize, which are useful for the ATCoEclipse system to incorporate existing features and invoke built-in functionalities to achieve multiple purposes such as capturing/replaying editing operations and retrieving source code regions and dependency relationships. Thirdly, Eclipse is widely used in the software industry, open source software development organizations and academic communities because it is free, open-source and extensible, which provides great opportunities for usability study and evaluations.

For supporting non-real-time collaboration functionalities, we have chosen the Subversion (SVN) version control system. Firstly, it is a sophisticated and popular system with a wide range of end-users. Secondly, it provides adequate interfaces for executing version control operations that can be utilized by external applications in communicating with the SVN system (e.g., sending version control commands). The SVN system is in-

corporated into the NRTCoS component (see Section 2.3.1) of the ATCoEclipse Server by invoking its public programming interfaces only, without modifying the source code.

For supporting advanced real-time collaboration functionalities, we seamlessly integrate both syntactic and semantic consistency maintenance techniques and incorporate them into the single-user Eclipse IDE without modifying its source code. In supporting the key DAL mechanisms, we invoke public programming interfaces of the built-in real-time source code syntax analysis functionalities provided by the Eclipse IDE for retrieving source code regions and dependency relationships (see Section 4.6.3). In this way, the correctness of the derivation can be ensured as it is provided by the IDE itself. In addition, the built-in syntax analysis functionalities embedded in the IDE are capable of meeting the real-time performance requirements for the DAL permission check and locking state update procedures.

By adopting the TA approach, each incorporated system or invoked technique can be replaced by other alternatives with similar functionalities and semantically compatible interfaces. For example, the SVN system can be replaced with the Git system, and the Eclipse IDE can be replaced with NetBeans⁴⁹ or Microsoft Visual Studio. In case of such replacement, only limited changes are needed in the ATCoEclipse components that directly interact with the replaced system/technique while all other components remain unchanged. Such characteristic makes the ATCoEclipse system customizable, and all approaches and techniques in building the ATCoEclipse system (as presented in this chapter) can also be applied to building other advanced collaboration supporting sys-

⁴⁹ <http://netbeans.org>

tems with similar functionalities and objectives.

6.3 The ATCoEclipse System

The ATCoEclipse system architecture presented in Figure 6.1 is an instance of the generic ATCoPE system architecture designed in Section 2.3. The *ATCoEclipse Client* is an instance of the ATCoPE Client, and the *ATCoEclipse Server* is an instance of the ATCoPE Server. Multiple ATCoEclipse Clients are connected to the ATCoEclipse Server via communication networks such as the Internet, and thus the ATCoEclipse system can be used by geographically dispersed software development teams and organizations.

Figure 6.1: The ATCoEclipse system architecture

6.3.1 The ATCoEclipse Server

To meet the requirements for high performance and scalability specified by Design Objective 4 in Section 2.2, the ATCoEclipse Server maintains minimum and lightweight server-side information and services that are necessary for realizing the ATCoPE func-

tionalties.

The NRTCoS component, which is designed for realizing non-real-time collaboration functionalities, only acts as a relay between the SVN system and the ATCoEclipse Clients. It simply passes the version control commands (e.g., *commit*, *update*) issued from the ATCoEclipse Clients to the SVN system, and transmits the data (e.g., source code copy) between the SVN repository and the ATCoEclipse Clients.

The real-time collaboration functionalities are implemented by three modules inside the RTCoS component of the server: (1) the *Group Management Module* is implemented for managing end-users' permissions in real-time collaboration sessions; (2) the *Cache Management Module* is implemented for maintaining the mapping between real-time collaboration sessions and corresponding data storages for the source code copies of the sessions; and (3) the *Session Management Module* is implemented for relaying communication messages (e.g., editing and locking operations) in real-time collaboration sessions, facilitating dynamic session membership management, and communicating with the NRTCoS component for realizing version control functionalities in real-time collaboration sessions.

6.3.2 The ATCoEclipse Client

The ATCoEclipse Client consists of the single-user Eclipse IDE for providing rich programming functionalities and the ATCoEclipse Client Adaptor for incorporating advanced collaboration capabilities. As presented in Figure 6.2, the ATCoEclipse Client Adaptor is implemented with five major functional modules, which invoke various programming interfaces provided by the underlying Eclipse IDE for realizing advanced real-time collaboration functionalities.

Figure 6.2: The ATCoEclipse Client system architecture and major functional modules

The *OT Module* is the key to supporting real-time collaboration functionalities and syntactic consistency maintenance features, with the OT technique as the cornerstone. The OT module makes use of the *generic collaborative engine* (GCE) contributed in prior work [98], which provides sophisticated OT-related functions with high efficiency.

The *DAL Module* implements all DAL-related functionalities and mechanisms for supporting semantic conflict prevention. It transparently invokes programming interfaces related to *Java Model* and *Abstract Syntax Tree* of the JDT Core⁵⁰ (embedded in the underlying Eclipse IDE) to efficiently retrieve the locations and scopes of the fields and

⁵⁰ <http://www.eclipse.org/jdt/core>

methods in Java classes and the method-field-reference and method-method-invocation relationships among them, which are required by the DAL mechanisms. It also invokes programming interfaces of the JDT UI⁵¹ to prohibit the end-user's editing action from execution whenever the editing permission is denied by the DAL permission check.

The *Awareness Module* implements novel collaboration awareness features for supporting the DAL scheme (to be presented in Section 6.4.2). It transparently invokes programming interfaces of the JDT Core to retrieve adequate information (e.g., scopes of locked regions) for supporting awareness mechanisms, and also utilizes programming interfaces of the JDT UI for displaying collaboration workspace and locking awareness information in the user interface of the source code editor.

The *Local Operation Handler* (LOH) module is responsible for capturing and propagating local editing operations. To capture local editing operations (the end-user's editing actions in the source code editor), the ATCoEclipse Client Adaptor registers a *listener* with the Eclipse Java Editor during the initialization procedure of the application. During the programming process, whenever an editing operation is issued by the local user, the listener captures the operation and retrieves its detailed information for DAL permission check. This procedure is executed exactly within the timeslot between the generation of the operation and its actual execution on the underlying source code document, which allows the DAL mechanism to prevent the issued editing operation from execution whenever the programmer's editing permission is denied by the DAL permission check. If the editing permission is granted, the operation is executed on the

⁵¹ <http://www.eclipse.org/jdt/ui>

underlying source code document, processed by the OT mechanism, and then propagated via the ATCoEclipse Server to all remote sites within the same session. Meanwhile, the DAL Module invokes the CFD locking state update procedure to update the locking states based on the evolved source code contents.

The *Remote Operation Handler* (ROH) is responsible for receiving and processing remote editing operations. For each remote operation received from the communication channel, it is first processed by the OT mechanism for syntactic consistency maintenance and replayed on the underlying source code document. The executed operation instantly becomes visible in the user interface of the source code editor due to the automatic mapping between the document and the view. Meanwhile, the CFD locking state update procedure is invoked to update the locking states based on the evolved source code contents.

The LOH and ROH modules are implemented as two parallel threads, which frequently invoke some common components. For example, both LOH and ROH threads will invoke the OT Module for processing editing operations. Similarly, the underlying JDT UI is also accessed by both LOH (for capturing local editing operations) and ROH (for replaying remote editing operations) threads. Since the procedure of handling an editing operation in either the LOH or ROH thread is an uninterruptible transaction, related techniques (e.g., semaphore, monitor) have been applied to control the parallel execution of LOH and ROH threads and serialize concurrent accesses to the OT Module, the JDT UI, and other shared components in the system.

6.4 User Interface Design of the ATCoEclipse Client

6.4.1 Major User Interfaces of the ATCoEclipse Client

Figure 6.3 illustrates the user interface of the ATCoEclipse Client for a programmer to initialize the collaborative programming work. The user interface is designed according to functional specifications in Sections 2.4.1, 2.4.2 and 2.4.3.

Figure 6.3: The user interface of the ATCoEclipse Client for initializing collaborative programming work

In the upper-left part of the dialog box, the programmer can specify the user account for using the SVN system, which is similar to the authentication process in using a conventional SVN client. Upon successful authentication (performed by the incorporated SVN system via the ATCoEclipse Server Interface), the trees of source code files and directories underneath the specified location are displayed in the lower-left part.

To create a new collaboration session, the programmer first browses the source code trees and specifies a location to check out, which acts similarly as a conventional SVN

client integrated in a single-user programming environment. Afterwards, the programmer proceeds to choose the collaboration mode for the session to be created in the right panel. As presented in Figure 6.3, there are two collaboration modes: non-real-time and real-time, which are designed according to specifications in Section 2.4.2.

In addition to creating a new session, the programmer can also join an existing real-time collaboration session, as specified in Section 2.4.3. The list of existing real-time collaboration sessions available for the programmer to join is shown in the lower part of the right panel in Figure 6.3. As displayed in the list, there are three real-time collaboration sessions available at the moment: (1) *RS-SimpleCalculator-Chengzheng*, which is a real-time collaboration session for the project named *SimpleCalculator* created by the programmer named *Chengzheng*; (2) *RS-AdvancedSearch-Yuqing*, which is a real-time collaboration session for a different project named *AdvancedSearch* created by another programmer named *Yuqing*; and (3) *RS-OTXplorer-Xuyi*, which is a real-time collaboration session for another project named *OTXplorer* created by the programmer named *Xuyi*.

Figure 6.4 illustrates an instance of the major user interface of the ATCoEclipse Client when a programmer is conducting the programming work in a real-time collaboration session. The *package explorer* shown in the left panel of the user interface looks similar to the one in the single-user Eclipse IDE, but it has been integrated with additional real-time collaboration features. When the programmer issues an editing operation in the package explorer to create, delete or rename source code files and directories, the operation is instantly propagated and executed at all other active sites in the session; conversely, changes on the source code tree from other active sites are also performed and noticed in this package explorer in real-time.

Figure 6.4: The major user interface of the ATCoEclipse Client

In the middle panel of the ATCoEclipse Client, the Java source code editor also looks similar to the single-user Eclipse Java source code editor, but it has been integrated with many advanced real-time collaboration functionalities and features.

Firstly, the local programmer's editing on the source code file is instantly propagated to other active sites of the same real-time file session, while editing operations generated from other active sites are also performed and noticed in real-time. While programmers

are concurrently editing the source code contents in a flexible and unconstrained fashion, the syntactic consistency of the shared source code copy is automatically maintained by the underlying OT technique at all times.

Secondly, the DAL mechanisms are also working automatically at the same time, without manual efforts from the programmer. The system checks the editing permission for each local operation and updates the locks granted to collaborating programmers based on the latest source code contents from time to time. If the programmer accidentally or intentionally issues an editing operation in a source code region that is currently locked by others, the editing action will be prevented from execution, and the programmer is notified of the permission deny in the user interface. Under such circumstances, the programmer may tentatively divert to another free region or open area, and later come back to this region if it has been unlocked by the collaborating programmer who was previously working in it.

In the right panel of the ATCoEclipse Client, collaboration-related information is displayed. In the upper part of the panel, the current collaboration mode (i.e., real-time or non-real-time), the session name, and the owner of the real-time collaboration session are displayed. In the middle part, the lists of active collaborators in the same real-time file session and the same real-time cluster session are displayed respectively, which are dynamically updated when a new site joins the session or an existing site leaves the session. In the lower part, the programmer may click the button to switch among different collaboration modes, and the SVN version control commands (i.e., *update* and *commit*) are also provided for collaborating programmers to use in real-time collaboration sessions. A notification box is placed below for delivering collaboration-related messages (e.g., who joins or leaves the session, who commits the latest source code copy to the

SVN repository on behalf of the group). The notification box is also integrated with instant chatting features for facilitating ad hoc communications during collaboration sessions.

6.4.2 The DAL-related Collaboration Workspace and Locking Awareness Feature in the ATCoEclipse User Interface

In multi-user collaborative applications, particularly the real-time collaborative editing systems, collaboration workspace awareness is an important and beneficial user interface feature for supporting distributed collaborators to learn where and what others are doing in the shared workspaces [49]. In the past studies, various workspace awareness features have been invented for supporting different application domains, such as *Multi-User Telepointer* and *Radar View* in *CoWord* [98][111] and *Multi-Perspective Radar-View*, *Televiewpointer* and *XPointer* in *CoMaya* [3][5][7].

In designing the user interface of the ATCoEclipse Client, we have contributed a novel collaboration awareness feature, which delivers collaboration workspace and locking awareness information in the presence of the DAL scheme, and complements the locking mechanism in collectively supporting semantic conflict prevention.

As aforementioned, the DAL locking mechanism prohibits concurrent work on selected source code regions for supporting semantic conflict prevention. If one programmer accidentally or intentionally issues an editing operation in a source code region that is currently locked by other programmers, the editing will be denied and the programmer will be notified. While the locking mechanism is applied to ensure the effectiveness of semantic conflict prevention, it is also equivalently important to keep collaborating programmers aware of the latest locking states on source code regions and the latest work-

spaces of other collaborators for the following two reasons.

Firstly, if a programmer is aware of the current locking states on source code regions, s/he can avoid editing source code regions locked by others, thus saving the time and effort wasted in being rejected while intentionally or accidentally issuing editing operations in the locked regions. Secondly, if a programmer is able to intuitively observe the entire picture (i.e., working regions and depended regions of all collaborators) of the collaborative work that is being carried out, s/he can better understand the relationships among programming tasks performed by different collaborators and pay more attention to the compatibility among individual tasks accordingly. In this regard, such kind of collaboration workspace and locking awareness is complementary to the locking mechanism in collectively supporting semantic conflict prevention, as it helps to reduce incompatible collaborative work and improve mutual understanding among programmers.

One principal objective of the DAL-related collaboration awareness feature is always keeping programmers aware of where and what other collaborators are doing during the real-time collaboration session. In addition to the workspace awareness, it is also necessary to indicate the depended regions with respect to the working region for each collaborator, since the depended regions are also prohibited from being edited by other collaborators under the DAL scheme. Moreover, the awareness information must be: (1) intuitive enough so that programmers can learn collaboration-related knowledge with minimal efforts, (2) seamlessly integrated with the source code editor's user interface rather than placed in a separated panel, in order to preserve the continuous programming work, and (3) served in a one-way fashion in the sense that the awareness mechanism works automatically without requiring programmers to response, thus avoiding disturbance to the programming work.

Guided by these objectives, the ATCoEclipse Client provides the DAL-related collaboration workspace and locking awareness feature by using distinctive background colors for highlighting different source code segments/regions as follows:

1. Unlocked segments of the source code file, including open areas and free regions, take the default background color (e.g., white);
2. Source code regions locked by a single programmer are highlighted by a unique color assigned to this programmer;
3. Among all locked regions for a programmer, the working region is further differentiated by a color bar displayed to the left of the vertical ruler that shows source code line numbers; and
4. Source code regions shared-locked by multiple programmers are highlighted by a special color to inform the collaborators of the lock-sharing.

Figure 6.5 illustrates the DAL-related collaboration workspace and locking awareness feature in the user interface of the ATCoEclipse Client. Within the lists of active collaborators in the right panel, a unique color has been assigned to each collaborator in the session, which is used for highlighting the working region and depended regions locked for this programmer in the source code editor. By observing the user interface of the source code editor in Figure 6.5, it can be intuitively learnt that the local programmer *Hongfei* is currently working on the *push* method with corresponding locks granted on this working region and its depended regions *int store[]*, *int max_length* and *int top*; and the remote collaborator *Chengzheng* is working on the *pop* method with a depended region *int top*. To differentiate the working region and depended regions of a programmer,

a color bar has been displayed to the left of each working region. Particularly, the field *int top* has been highlighted by the special color (green) to indicate that it is currently shared-locked by both programmers.

Figure 6.5: DAL-related collaboration workspace and locking awareness feature in ATCoEclipse Client user interface

Apart from these colored regions, other segments of the source code are currently uncolored (white), so they are unlocked and available for anyone to access. With these awareness features seamlessly integrated with the user interface, programmers can intuitively learn the latest locking states on various source code regions and understand the relationships among collaborators' programming tasks.

6.5 Preliminary Performance Feedback and Analysis

The successful implementation of the ATCoEclipse prototype system has validated the feasibility of various approaches, techniques and solutions derived in this research, and extended the TA approach for building any-time collaborative programming systems. The ATCoPE architectures and technical solutions, together with the prototype implementation, have collectively achieved the functional design objectives (Design Objectives 1, 2 and 3) proposed in Section 2.2, which are briefly summarized as follows.

Table 6.1: Summary of functional design objectives and technical solutions

Design Objective	Technical Solution
Design Objective 1	<ol style="list-style-type: none"> 1. Transparently incorporating existing single-user IDEs 2. Transparently incorporating existing version control systems
Design Objective 2	<ol style="list-style-type: none"> 1. Achieving unconstrained real-time collaboration with consistency maintenance by conflict resolution 2. Supporting semantic conflict prevention by DAL 3. Providing DAL-related collaboration workspace and locking awareness feature in the client user interface
Design Objective 3	<ol style="list-style-type: none"> 1. Supporting the management of multiple simultaneous active collaboration sessions in the server 2. Achieving dynamic session membership management for real-time collaboration sessions 3. Bridging non-real-time collaboration services and real-time collaboration sessions 4. Supporting convenient transitions among collaboration sessions and modes in the client user interface

In this section, we present preliminary performance feedback and analysis, including a set of micro benchmark experiments on essential algorithms and procedures, to confirm that the ATCoEclipse prototype has achieved good system performance and scalability as specified by Design Objective 4 in Section 2.2.

6.5.1 General Feedback and Analysis

Firstly, the local responsiveness of the ATCoEclipse Client is as good as the single-user Eclipse IDE for the following reasons: (1) as the shared source code copy is replicated at all collaborating sites in a real-time collaboration session, local editing operations can be executed instantly without communication with the server or remote sites; (2) the DAL permission check procedure is executed by consulting the local locking state data structures only; and (3) the locking state update procedure is also executed based on the local source code contents and contextualized editing positions only.

Secondly, the notification and integration of remote editing operations are performed in real-time for the following reasons: (1) the propagation of editing operations to remote sites can be completed instantly because most communication messages are less than 1kB in size, which can reach inter-continental remote sites normally within a delay of 100ms; (2) there is no DAL permission check needed for a remote editing operation under the shared-locking scheme; and (3) the locking state update procedure executed with respect to a remote editing operation is performed based on the local source code contents and contextualized editing positions only, which is exactly the same as the locking state update for a local editing operation.

Thirdly, in terms of the dynamic session membership management for real-time collaboration sessions, a global synchronization procedure (i.e., the join-protocol procedure

for accepting new members in real-time collaboration sessions) involves a maximum of three single-trip messages (as designed in Section 3.3), which can be completed within a total timeslot of less than 300ms (100ms for each single-trip). The transmission of the entire source code copy of the session can be costly depending on the total size of the source code files, but such delay only takes place at the new client but not at existing clients of the session because the source code copy is directly retrieved from the RTCoS source code cache at the server, without involving other existing clients.

Fourthly, the ATCoEclipse system is scalable for accommodating a large number of active collaborating programmers and real-time collaboration sessions, which has been enabled by the following major factors: (1) for supporting non-real-time collaboration functionalities, the ATCoEclipse Server is never involved in maintaining information or providing services, but solely relies on the incorporated SVN system; and (2) for supporting real-time collaboration functionalities, the ATCoEclipse Server only maintains minimum lightweight server-side information (such as real-time session membership) and services (such as message relaying), whereas most heavy-duty tasks (such as operation capturing/replaying, OT, permission check and locking state update) are fully completed at individual and separated ATCoEclipse Clients. In addition, the replicated architecture and the shared-locking scheme are two minor factors for supporting good scalability, because the responsiveness of local operations will never be affected by the increase of the number of active collaborating sites within the session.

6.5.2 Micro Benchmark Experiments

In addition to the general feedback and analysis, we have designed and conducted a set of micro benchmark experiments on several essential algorithms and procedures that are

critical to the system performance. As the system has been implemented by transparently integrating existing systems with extra functionalities and features contributed by this research, we focus on investigating how additional functionalities affect the performance of the original system and how many extra overheads have been incurred.

For generating the input data (i.e., source code documents) for the micro benchmark experiments, we have designed and implemented a *source code generator*, which can generate Java source code documents according to a set of customizable parameters, including the amount of fields and methods in a Java class and the total amount of field references and method invocations as well, which are important factors in determining the runtime overheads (as discussed in Section 4.7 and Section 5.4). Given multiple sets of source code documents generated with different parameters, the micro benchmark experiments have been conducted. The experimental platform is a workstation with Intel Core i7-3770 CPU @ 3.40GHz and 4.00 GB of RAM.

Firstly, we measure the performance of the two utility functions related to source code contents and the dependency graph, namely *GetWorkingRegion* and *GetDepRegionSet*, which have been presented in Section 4.7.1. As the runtime overheads of the two functions depend on the amount of basic regions (denoted as N) and the total amount of field references and method invocations (denoted as M), 5 sets of benchmark experiments have been conducted with source code documents having different N and M values. The results of the performance measurement are presented in Table 6.2.

This set of experimental results has confirmed the good performance of both *GetWorkingRegion* and *GetDepRegionSet* functions. As presented, the runtime overheads grow steadily with the increase of the N and M values. Even for the source code document

containing 800 fields and methods with 32000 field references and method invocations (which is unrealistic), the *GetWorkingRegion* function costs only 0.0005021ms and the *GetDepRegionSet* function costs only 1.9072ms. This set of experiments also confirmed the significant advantage of the implicit derivation approach proposed in Section 4.6.3.

Table 6.2: Results of performance measurement on utility functions related to source code contents and the dependency graph

Unit: millisecond

Utility Function	N = 50 M = 2000	N = 100 M = 4000	N = 200 M = 8000	N = 400 M = 16000	N = 800 M = 32000
<i>GetWorkingRegion</i>	0.0000231	0.0000480	0.0000968	0.0002114	0.0005021
<i>GetDepRegionSet</i>	0.0445	0.0956	0.2512	0.7570	1.9072

N: amount of basic regions (including fields and methods)

M: total amount of field references and method invocations

Secondly, we measure the performance of the three utility functions related to locking state data structures, namely *CheckOwnership*, *GrantLocks* and *ReleaseLocks*, which have been initially presented in Section 4.7 and redesigned in Section 5.4.1 for the extended locking state data structures. In addition to the amount of basic regions (denoted as N) and the total amount of field references and method invocations (denoted as M), the runtime overheads of these three functions also depend on the number of collaborating users (denoted as U) in the real-time file session. Therefore, 20 sets of benchmark experiments have been conducted with source code documents having different N and M values and real-time file sessions having different U values. The results of the performance measurement are presented in Table 6.3.

The experimental results have confirmed the good performance of the three utility functions. As presented, the runtime overhead of each function grows slowly with the in-

crease of either the size of the source code document or the number of collaborating users. Even for the worst case where the real-time file session involves 16 collaborating users and the source code document contains 800 fields and methods with 32000 field references and method invocations (which is unrealistic), the *CheckOwnership* function costs only $0.00890 * 10^{-3}$ ms while the *GrantLocks* and *ReleaseLocks* functions cost only $9.916 * 10^{-3}$ ms and $6.650 * 10^{-3}$ ms respectively, which are extremely efficient. This set of experimental results has confirmed that the locking state data structures and related functions have been designed appropriately under the current circumstances.

Table 6.3: Results of performance measurement on utility functions related to locking state data structures

Unit: 10^{-3} millisecond

U	Utility Function	N = 50 M = 2000	N = 100 M = 4000	N = 200 M = 8000	N = 400 M = 16000	N = 800 M = 32000
2	<i>CheckOwnership</i>	0.00316	0.00337	0.00356	0.00372	0.00439
	<i>GrantLocks</i>	0.186	0.422	1.156	3.187	7.923
	<i>ReleaseLocks</i>	0.146	0.308	0.688	1.502	2.919
4	<i>CheckOwnership</i>	0.00357	0.00376	0.00405	0.00435	0.00514
	<i>GrantLocks</i>	0.188	0.432	1.212	3.231	8.407
	<i>ReleaseLocks</i>	0.166	0.352	0.842	1.859	3.469
8	<i>CheckOwnership</i>	0.00424	0.00464	0.00483	0.00528	0.00634
	<i>GrantLocks</i>	0.190	0.436	1.280	3.298	9.152
	<i>ReleaseLocks</i>	0.216	0.444	1.138	2.460	4.405
16	<i>CheckOwnership</i>	0.00562	0.00610	0.00663	0.00748	0.00890
	<i>GrantLocks</i>	0.198	0.453	1.402	3.631	9.916
	<i>ReleaseLocks</i>	0.303	0.623	1.714	3.853	6.650

N: amount of basic regions (including fields and methods)

M: total amount of field references and method invocations

U: number of collaborating users in the session

Finally, we measure the performance of the two essential procedures, namely *PermissionCheck* and *CFD_LSUpdate*, which have integrated all utility functions to realize the

fundamental DAL mechanisms. The overall runtime overheads of the two procedures have been measured by 20 sets of benchmark experiments with different source code documents and real-time file sessions, and the results of the performance measurement are presented in Table 6.4.

Table 6.4: Results of performance measurement on essential procedures

Unit: millisecond

U	Procedure	N = 50 M = 2000	N = 100 M = 4000	N = 200 M = 8000	N = 400 M = 16000	N = 800 M = 32000
2	<i>PermissionCheck</i>	0.0000287	0.0000553	0.0001091	0.0002233	0.0005182
	<i>CFD_LSUpdate</i>	0.0864	0.1909	0.5249	1.5122	3.7331
4	<i>PermissionCheck</i>	0.0000303	0.0000573	0.0001139	0.0002283	0.0005294
	<i>CFD_LSUpdate</i>	0.1762	0.3732	1.0140	3.0132	7.4968
8	<i>PermissionCheck</i>	0.0000317	0.0000581	0.0001151	0.0002315	0.0005375
	<i>CFD_LSUpdate</i>	0.3503	0.7437	2.0438	6.2269	15.4252
16	<i>PermissionCheck</i>	0.0000329	0.0000597	0.0001163	0.0002387	0.0005606
	<i>CFD_LSUpdate</i>	0.6967	1.4959	4.1162	12.4635	30.6650

N: amount of basic regions (including fields and methods)

M: total amount of field references and method invocations

U: number of collaborating users in the session

The experimental results in Table 6.4 have confirmed the good overall performance of the DAL mechanisms implemented in the ATCoEclipse prototype system. As presented, the runtime overheads of both procedures grow steadily with the increase of the size of the source code document and the number of collaborating users. Particularly, the cost of the *PermissionCheck* procedure grows quite slowly with the increase of the number of collaborating users. Even for the worst case in the table (which is unrealistic), the *PermissionCheck* procedure costs only 0.0005606ms and the *CFD_LSUpdate* procedure costs only 30.6650ms. It can therefore be confirmed that the DAL mechanisms within the ATCoEclipse system, which have been additionally incorporated on top of the ex-

isting single-user programming system (i.e., Eclipse IDE), have brought very low overheads while providing extra functionalities. As discussed in Section 6.5.1, the efficient executions of these two essential procedures are critical in meeting the Design Objective 4. The low costs of *PermissionCheck* and *CFD_LSUpdate* contribute to the high local responsiveness, while the low cost of *CFD_LSUpdate* also contributes to the fast remote notification. Furthermore, the good performance of the two procedures also supports the good system scalability, as the runtime overheads grow steadily and slowly with the increase of the number of collaborating users. In conclusion, the benchmark experimental results have confirmed the good system performance from a quantitative perspective.

6.6 Summary

In this chapter, we presented the design and implementation of the ATCoEclipse prototype system, and contributed a package of techniques for building ATCoPE systems.

Firstly, we designed and implemented the ATCoEclipse system under the transparent adaptation (TA) approach, which has been extended in this research by (1) integrating existing single-user and non-real-time collaborative programming techniques and systems with novel real-time collaborative programming techniques and (2) integrating both syntactic consistency maintenance and semantic conflict prevention techniques for supporting real-time collaboration. The ATCoEclipse Server has been designed for maintaining minimum and lightweight server-side information and services necessary for the real-time collaboration, whereas non-real-time collaboration functionalities have been realized by transparently incorporating the SVN version control system. The ATCoEclipse Client has transparently converted the single-user Eclipse IDE into a multi-user collaborative IDE by means of the ATCoEclipse Client Adaptor, which transpar-

ently invokes various programming interfaces of the underlying Eclipse IDE for realizing advanced collaboration functionalities, including both syntactic and semantic consistency maintenance features that have been seamlessly integrated.

Secondly, we presented the design of major user interfaces of the ATCoEclipse Client, which is compatible with the single-user Eclipse IDE while providing extra user interface features for advanced collaboration. In designing the user interface, we also contributed a novel DAL-related collaboration workspace and locking awareness feature, which not only provides the locking awareness information in the presence of the DAL scheme, but also complements the locking mechanism in collectively supporting semantic conflict prevention.

Thirdly, we presented a set of preliminary performance feedback and analysis, which have confirmed that the ATCoEclipse prototype achieves good system performance and scalability with respect to the design objective. The performance analysis has addressed several major aspects, including the high local responsiveness of the ATCoEclipse Client, the real-time notification and integration of remote editing operations, the efficient processing of the join-protocol procedure for dynamic session membership management, and the good system scalability for accommodating a large number of active collaborating sites and real-time collaboration sessions. In addition, we conducted a set of micro benchmark experiments on essential algorithms and procedures, and the results have confirmed the good system performance from a quantitative perspective.

In the next chapter, we will conclude this research, summarize its major contributions, and discuss several research directions and issues identified for the future work.

Chapter 7.

Conclusions and Future Work

7.1 Conclusions

Collaboration is needed in programming due to the requirements for diverse expertise and skills to solve complex problems and for producing increasingly large and complex software systems within tight schedules imposed by competitive software markets. Past research has invented two collaborative programming paradigms, namely non-real-time collaborative programming and real-time collaborative programming, with different characteristics and properties, complementary needs and applications, and respective supporting techniques. This research has proposed a novel collaborative programming paradigm named *any-time collaborative programming*, which aims to seamlessly integrate both real-time and non-real-time collaborative programming to meet the complementary and dynamic collaboration needs in programming processes. Under the any-time collaborative programming paradigm, multiple programmers may work in real-time and/or non-real-time collaboration modes supported by the most suitable techniques and tools, and flexibly switch among different collaboration modes as new collaboration needs arise dynamically during the programming process. This research has focused on the architecture and enabling techniques for supporting and realizing any-time collaborative programming, and made important contributions in three major areas as summarized in the following sub-sections.

7.1.1 Any-Time Collaborative Programming Environment (ATCoPE)

This research has contributed an *Any-Time Collaborative Programming Environment*

(ATCoPE), which converts the abstract notion of any-time collaborative programming into concrete system architectures and functionalities. ATCoPE also serves as a general framework for investigation and experimentation with any-time collaboration enabling techniques throughout this research. Major contributions in this area include:

1. The specifications of a set of general design objectives, which are not only applicable to ATCoPE, but also generic and suitable for other collaboration supporting systems and environments with similar functionalities;
2. The design of the system architecture and major functional components of ATCoPE, which have realized the notion of any-time collaborative programming and established a high-level framework for the design and implementation in lower-levels;
3. The specifications of the working process and major functionalities of the ATCoPE system from the end-users' perspective, which allow collaborating programmers to use similar user interfaces and compatible functionalities for both non-real-time and real-time collaboration in one integrated environment, and enjoy any-time collaboration functionalities in terms of the flexible transitions among different collaboration modes and sessions and the interactions among non-real-time collaborating programmers and real-time collaborating groups;
4. Two sets of syntactic consistency maintenance techniques devised for supporting real-time collaborative programming, one for real-time cluster sessions and the other for real-time file sessions;
5. The distributed join-protocol and leave-protocol devised for supporting dynamic session membership management in real-time collaboration sessions, which are key

techniques for enabling flexible and smooth transitions among different collaboration sessions under the any-time collaborative programming paradigm; and

6. The design of the mechanisms for supporting non-real-time collaboration functionalities in both non-real-time and real-time collaboration sessions, which have enabled the interactions among non-real-time collaborating programmers and real-time collaborating groups under the any-time collaborative programming paradigm.

7.1.2 Dependency-based Automatic Locking (DAL)

This research has contributed a package of novel techniques for supporting semantic conflict prevention in real-time collaborative programming under the ATCoPE framework. Major contributions in this area include:

1. The exploration and in-depth analysis on representative programming scenarios for understanding and deriving the nature and general conditions of semantic conflicts, which have motivated the *Dependency-based Automatic Locking* (DAL) approach;
2. The DAL approach for semantic conflict prevention, which is capable of (1) supporting automatic, responsive and fine-grained locking on selected source code regions and (2) balancing conflict prevention, concurrent work and programmer convenience in real-time collaborative programming sessions;
3. The formal definitions of the basic region, the dependency relationship and the dependency graph (DG), which are key elements of the DAL approach and the foundation for the design of DAL mechanisms and schemes;
4. The conditions and algorithms for the DAL permission check, which is the key to

realizing the DAL mechanism in semantic conflict prevention;

5. The implicit derivation approach and techniques for efficient maintenance of the DG and locking state, which are important elements related to the technical feasibility of the DAL approach;
6. The shared-locking scheme, which allows three different types of shared-locking under well-defined conditions, and supports semantic conflict prevention in unconstrained real-time collaborative programming where programmers are allowed to flexibly perform concurrent and/or dynamic DG editing operations; and
7. The contextualization and full derivation (CFD) scheme and techniques, which are capable of supporting consistent and correct locking state update under the extended DAL scheme.

7.1.3 The ATCoEclipse Prototype System

This research has contributed the design and implementation of the *ATCoEclipse* prototype system, which serves as a proof-of-concept of ATCoPE and an experimental research vehicle for exploration and investigation of ATCoPE system building issues and techniques. Major contributions in this area include:

1. The extensions to the TA approach for building any-time collaborative programming systems by seamlessly integrating existing single-user and non-real-time collaborative programming systems with novel real-time collaborative programming techniques which support both syntactic and semantic consistency maintenance features, without changing the source code of existing systems;

2. The design and implementation of the ATCoEclipse prototype system, which has realized and validated all major approaches, techniques and solutions derived in this research, and provided positive confirmation of their feasibility and preliminary performance feedback;
3. The design of major user interfaces of the ATCoEclipse Client, which preserves existing user interfaces and seamlessly integrates them with extra features for novel collaboration functionalities, thus enabling programmers to use familiar interfaces for conventional functionalities while enjoying novel features at the same time; and
4. The DAL-related collaboration workspace and locking awareness feature integrated with the user interface, which not only supports locking awareness in the presence of the DAL scheme, but also complements the locking mechanism for collectively supporting semantic conflict prevention.

7.2 Future Work

This research has contributed enabling techniques for supporting any-time collaborative programming in three areas as presented above, and advanced the supporting technologies for collaborative software development. The research has established key foundations and general frameworks for exploration, investigation and development in a range of research areas and directions for supporting advanced collaboration functionalities in software development, and provided the potentials for more innovations in several research domains including software engineering, computer-supported cooperative work (CSCW), human-computer interaction (HCI), and distributed computing. Throughout the research process, several research directions and issues have been identified for fu-

ture exploration and investigation.

7.2.1 ATCoPE Supporting Techniques

Different from many real-time collaborative editing systems, the real-time collaboration functionalities of ATCoPE support real-time propagation of both file-level and cluster-level editing operations. In devising syntactic consistency maintenance techniques for cluster-level editing operations, we have adopted the optimistic conflict resolution approach for supporting responsive, unconstrained and concurrent execution of cluster-level editing operations, derived a comprehensive set of conflict/compatible relationships between all pairs of cluster-level editing operations, and specified the resolution results for all pairs of conflicting operations. More exploration and investigation are needed in this direction to generalize the solutions and invent a novel OT technique that is capable of supporting and transforming any cluster-level editing operation defined on any hierarchical tree structure that consists of files and directories. The technique would be generic, which not only supports collaborative programming, but also contributes to a wide range of collaborative environments where collaborating users deal with a package of hierarchically-related working artifacts rather than a single document.

ATCoPE supports the use of non-real-time version control functionalities within real-time collaboration sessions, thus enabling the interactions among non-real-time and real-time collaborating programmers, which serves as a unique feature of any-time collaborative programming. This research has adopted two common operations, namely *commit* and *update*, for the proof-of-concept design of the working mechanisms. Follow-up investigation will be conducted for supporting more version control operations in real-time collaboration sessions, and will focus on how the system can better facili-

tate the use of those operations from the end-users' perspective. In addition, the processing of a *commit* or *update* version control command may produce conflicts and lead to the conflict resolution, which creates another research issue on how ATCoPE can assist real-time collaborating programmers in the conflict resolution process in terms of functional support and user interface features.

At the beginning of this thesis, we illustrated a sample programming process to demonstrate the dynamic collaboration needs, which was used for motivating the notion of any-time collaborative programming. Based on the well-justified design of the system architecture and major components, ATCoPE is a generic framework that has the potential for supporting any kind of collaborative programming with any type of interactions among programmers, as well as supporting other software engineers (e.g., designers) to collaborate in other software development activities (e.g., software design). More investigation will be conducted to further generalize the ATCoPE framework for supporting multiple software development activities and phases. In addition, as there is a variety of emerging software development methods (such as agile software development) that attract increasing interests from the software industry, we will explore how ATCoPE can play an important role in supporting advanced collaboration under different software development methods and models. Ultimately, we would also plan to apply the ATCoPE framework and solutions to other collaborative environments, because the any-time collaboration needs derived from this research may also exist in any other domain where collaboration is needed, and all approaches, techniques and solutions for ATCoPE are widely applicable as none of them necessarily binds to the programming domain. The wide applicability is one of the most important values of ATCoPE, which will be demonstrated through more experimental research work in the future.

7.2.2 DAL Supporting Techniques

In supporting semantic conflict prevention, various DAL schemes and mechanisms have been designed for balancing conflict prevention and concurrent work. Our ongoing research has already worked out some preliminary solutions to provide more flexible options in balancing conflict prevention and concurrent work, including a *flexible locking depth* scheme that allows the adjustment of the number of dependency levels within a locking scope, and a *multiple locking range* scheme that allows individual programmers to hold locks on multiple working regions at the same time. In the future work, we will further extend these solutions and contribute a systematic package as another important extension to the current DAL scheme. In addition, more in-depth investigation will be conducted to study the relationship between conflict prevention and concurrent work in real-world programming scenarios, and to explore whether and how DAL mechanisms may affect programmers' way of working and collaboration in software development.

As aforementioned, the current DAL mechanisms and techniques have focused on supporting semantic conflict prevention with regards to the concurrent programming work in a single source code file, but all techniques devised in this research are generic and applicable to semantic conflict prevention in a larger scope of source code files. Our future work will extend the current DAL scheme for supporting cross-file semantic conflict prevention in the programming work, which will start from investigating the dependency relationships among programming elements over multiple source code files and packages. This piece of extension work will make contributions on cross-file source code dependency modeling, and testify the generality of the DAL mechanism.

In addition to extending the DAL scheme and techniques within the domain of collabo-

rative programming, another promising research direction is to apply the DAL approach and techniques in other application domains such as collaborative document editing and collaborative media design. Investigation in this direction will further increase the value of the DAL approach and techniques, and make important contributions on the dependency modeling of document elements in different domains. Moreover, our future work will also explore the potential to utilize dependency relationships for supporting other purposes rather than semantic conflict prevention.

7.2.3 ATCoPE System Building Techniques and Usability Study

For implementing the ATCoEclipse system, we have incorporated the SVN system and the Eclipse IDE. One possible direction for the future work is to further practice the TA approach in building other ATCoPE systems by incorporating alternative systems and techniques, which will testify the generality of the system building techniques, and expand the end-user communities that may benefit from ATCoPE techniques. Moreover, throughout the design and implementation of multiple similar ATCoPE systems, more common issues involved in ATCoPE system building can be explored and investigated, such as the API requirements imposed on the incorporated systems.

In terms of the user interface design, research efforts will be made to contribute more innovative collaboration awareness features in ATCoPE, and to investigate the impacts of different awareness features on the way of working, the style of collaboration, the quality of the source code produced, and the productivity of collaborating programmers. In the presence of the DAL scheme, more user interface features will be contributed to better assist collaborating programmers in semantic conflict prevention.

Last but not least, the ATCoEclipse prototype system will be continuously developed

and improved for conducting usability evaluation and study. The usability study will be conducted with real-world collaborative programming scenarios and tasks which may involve multiple programming teams with different professional levels, different sizes, and diverse collaboration styles. The study will explore several major and generic issues including: (1) whether the ATCoPE working process is easy to follow; (2) whether programmers are able to conduct traditional single-user and non-real-time collaborative programming without learning new knowledge; and (3) whether the additional functionalities and features for real-time and any-time collaborative programming can effectively and efficiently facilitate the diverse and dynamic collaboration needs, and whether the new techniques have brought benefits and convenience to the programming work rather than disturbance and damage. The study will also address a wide range of detailed usability issues, including but not limited to: (1) how conflict resolution approaches and mechanisms may affect the programmers; (2) whether and how semantic conflict prevention approaches and mechanisms may benefit and affect the programmers and improve the quality/productivity of real-time collaboration, and whether there is any drawback, constraint or inconvenience; and (3) whether the user interface design and awareness features are appropriate and beneficial to the end-users, whether the notification messages and awareness information are accurate, adequate and understandable, and whether there is any adverse impact incurred. In addition, more fine-grained evaluations on system functionalities, performance and scalability will be conducted together with the usability study, which would collectively serve as the guidance for further extensions and improvement of the ATCoEclipse system as well as the generic ATCoPE supporting techniques.

Bibliography

- [1] Abdel-Wahab, H. M. and Feit, M. A. XTV: a framework for sharing X Window clients in remote synchronous collaboration. In *Proc. of TRICOMM* (1991), 159-167.
- [2] Ackerman, M. S. and Mcdonald, D. W. Collaborative Support for Informal Information in Collective Memory Systems. *Information Systems Frontiers* 2, 3-4 (2000), 333-347.
- [3] Agustina. Real-Time Collaborative Technologies and Systems for 3D Digital Media Design over the Internet. *PhD Thesis* (Nanyang Technological University, 2012).
- [4] Agustina, Liu, F., Xia, S., Shen, H., and Sun, C. CoMaya: incorporating advanced collaboration capabilities into 3d digital media design tools. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2008), 5-8.
- [5] Agustina and Sun, C. Televiewpointer: an integrated workspace awareness widget for real-time collaborative 3d design systems. In *Proc. of ACM Intl. Conf. on Supporting Group Work* (2010), 21-30.
- [6] Agustina and Sun, C. Dependency-conflict detection in real-time collaborative 3D design systems. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2013), 715-728.
- [7] Agustina and Sun, C. Xpointer: an x-ray telepointer for relaxed-space-time wysiwis and unconstrained collaborative 3d design systems. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2013), 729-740.
- [8] Agustina, Sun, C., and Xu, D. Operational transformation for dependency conflict resolution in real-time collaborative 3D design systems. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2012), 1401-1410.
- [9] Allen, L., Fernandez, G., Kane, K., Leblang, D. B., Minard, D., and Posner, J.

ClearCase MultiSite: Supporting Geographically-Distributed Software Development. In *Software Configuration Management: Selected Papers from ICSE SCM-4 and SCM-5 Workshops, LNCS 1005* (1995), 194-214.

- [10] Baecker, R. M., Nastos, D., Posner, I. R., and Mawby, K. L. The user-centered iterative design of collaborative writing software. In *Proc. of INTERACT'93 and CHI'93 Conf. on Human Factors in Computing Systems* (1993), 399-405.
- [11] Bar, M. and Fogel, K. *Open Source Development with CVS (3rd Ed.)*. Paraglyph Press (2003).
- [12] Beck, K. Embracing Change with Extreme Programming. *Computer* 32, 10 (1999), 70-77.
- [13] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley (1999).
- [14] Berliner, B. CVS II: Parallelizing Software Development. In *Proc. of USENIX Winter 1990 Technical Conf.* (1990), 341-352.
- [15] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley (1987).
- [16] Blackburn, J. D., Hoedemaker, G., and Van Wassenhove, L. N. Concurrent software engineering: prospects and pitfalls. *IEEE Transactions on Engineering Management* 43, 2 (1996), 179-188.
- [17] Blackburn, J. D., Scudder, G. D., and Van Wassenhove, L. N. Improving Speed and Productivity of Software Development: A Global Survey of Software Developers. *IEEE Trans. Softw. Eng.* 22, 12 (1996), 875-885.
- [18] Blackburn, J., Scudder, G., and Van Wassenhove, L. N. Concurrent software development. *Commun. ACM* 43, 11es, Article 4 (2000).
- [19] Boehm, B., Bose, P., Horowitz, E., and Lee, M. J. Software requirements negotiation and renegotiation aids. In *Proc. of Intl. Conf. on Software Engineering* (1995), 243-253.

- [20] Boehm, B. and Egyed, A. Software requirements negotiation: some lessons learned. In *Proc. of Intl. Conf. on Software Engineering* (1998), 503-506.
- [21] Boyer, K. E., Dwight, A. A., Fondren, R. T., Vouk, M. A., and Lester, J. C. A development environment for distributed synchronous collaborative programming. In *Proc. of Annual Conf. on Innovation and Technology in Computer Science Education* (2008), 158-162.
- [22] Brooks, F. P. No Silver Bullet—Essence and Accidents of Software Engineering. *Computer* 20, 4 (1987), 10-19.
- [23] Brooks, F. P. *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley (1995).
- [24] Buffenbarger, J. Syntactic Software Merging. In *Selected papers from the ICSE SCM-4 and SCM-5 Workshops on Software Configuration Management* (1995), 153-172.
- [25] Cataldo, M. and Herbsleb, J. D. Communication patterns in geographically distributed software development and engineers' contributions to the development effort. In *Proc. of Intl. Workshop on Cooperative and Human Aspects of Software Engineering* (2008), 25-28.
- [26] Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., and Carley, K. M. Identification of coordination requirements: implications for the Design of collaboration and awareness tools. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2006), 353-362.
- [27] Chacon, S. *Pro Git (1st ed.)*. Apress (2009).
- [28] Cockburn, A. and Williams, L. The costs and benefits of pair programming. In *Extreme Programming Examined*, Addison-Wesley (2001), 223-243.
- [29] Collins-Sussman, B. The subversion project: buiding a better CVS. *Linux J.* 2002, 94 (2002), 3.
- [30] Conradi, R. and Westfechtel, B. Version models for software configuration

management. *ACM Comput. Surv.* 30, 2 (1998), 232-282.

- [31] Čubranić, D. and Storey, M. A. D. Collaboration support for novice team programming. In *Proc. of Intl. ACM SIGGROUP Conference on Supporting Group Work* (2005), 136-139.
- [32] Curtis, B., Krasner, H., and Iscoe, N. A field study of the software design process for large systems. *Commun. ACM* 31, 11 (1988), 1268-1287.
- [33] De Souza, C. R. B., Redmiles, D., Cheng, L.-T., Millen, D., and Patterson, J. How a good software practice thwarts collaboration: the multiple roles of APIs in software development. *SIGSOFT Softw. Eng. Notes* 29, 6 (2004), 221-230.
- [34] Ellis, C. A. and Gibbs, S. J. Concurrency control in groupware systems. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data* (1989), 399-407.
- [35] Ellis, C. A., Gibbs, S. J., and Rein, G. Groupware: some issues and experiences. *Commun. ACM* 34, 1 (1991), 39-58.
- [36] Estublier, J. Software configuration management: a roadmap. In *Proc. of Conf. on the Future of Software Engineering* (2000), 279-289.
- [37] Fan, H. and Sun, C. Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming. In *Proc. of ACM Symposium on Applied Computing* (2012), 737-742.
- [38] Fan, H. and Sun, C. Achieving integrated consistency maintenance and awareness in real-time collaborative programming environments: the CoEclipse approach. In *Proc. of IEEE Intl. Conf. on Computer Supported Cooperative Work in Design* (2012), 94-101.
- [39] Fan, H. and Sun, C. Supporting semantic conflict prevention in real-time collaborative programming environments. *ACM SIGAPP Applied Computing Review* 12, 2 (2012), 39-52.
- [40] Fan, H., Sun, C., and Shen, H. ATCoPE: any-time collaborative programming environment for seamless integration of real-time and non-real-time teamwork in

- software development. In *Proc. of ACM Intl. Conf. on Supporting Group Work* (2012), 107-116.
- [41] Flor., N. V. Side-by-side collaboration: a case study. *Int. J. Hum.-Comput. Stud.* 49, 3 (1998), 201-222.
 - [42] Flor., N. V. Globally distributed software development and pair programming. *Commun. ACM* 49, 10 (2006), 57-58.
 - [43] Frost, R. Jazz and the eclipse way of collaboration. *IEEE Softw.* 24, 6 (2007), 114-117.
 - [44] Garfinkel, D., Welti, B. C., and Yip, T. W. HP SharedX: A Tool for Real-Time Collaboration. *HP Journal* 45, 2 (1994), 23-36.
 - [45] Goldman, M., Little, G., and Miller, R. C. Real-time collaborative coding in a web IDE. In *Proc. of ACM Symposium on User interface Software and Technology* (2011), 155-164.
 - [46] Greenberg, S. and Marwood, D. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (1994), 207-217.
 - [47] Grinter, R. E. Systems architecture: product designing and social engineering. In *Proc. of Intl. Joint Conf. on Work Activities Coordination and Collaboration* (1999), 11-18.
 - [48] Grinter, R. E., Herbsleb, J. D., and Perry, D. E. The geography of coordination: dealing with distance in R&D work. In *Proc. of Intl. ACM SIGGROUP Conf. on Supporting Group Work* (1999), 306-315.
 - [49] Gutwin, C. and Greenberg, S. The effects of workspace awareness support on the usability of real-time distributed groupware. *ACM Trans. Comput.-Hum. Interact.* 6, 3 (1999), 243-281.
 - [50] Herbsleb, J. D. Global Software Engineering: The Future of Socio-technical Coordination. In *2007 Future of Software Engineering* (2007), 188-198.

- [51] Herbsleb, J. D. and Mockus, A. An Empirical Study of Speed and Communication in Globally Distributed Software Development. *IEEE Trans. Softw. Eng.* 29, 6 (2003), 481-494.
- [52] Hickey, T. J., Langton, J., and Alterman, R. Enhancing CS programming lab courses using collaborative editors. *J. Comput. Small Coll.* 20, 3 (2005), 157-167.
- [53] Ho, C., Raha, S., Gehringer, E., and Williams, L. Sangam: a distributed pair programming plug-in for Eclipse. In *Proc. of OOPSLA Workshop on Eclipse Technology Exchange* (2004), 73-77.
- [54] Knister, M. and Prakash, A. Issues in the design of a toolkit for supporting multiple group editors. *Computing Systems* 6, 2 (1993), 135-166.
- [55] Kraut, R. E. and Streeter, L. A. Coordination in software development. *Commun. ACM* 38, 3 (1995), 69-81.
- [56] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558-565.
- [57] Langton, J. T., Hickey, T. J., and Alterman, R. Integrating tools and resources: a case study in building educational groupware for collaborative programming. *J. Comput. Sci. Coll.* 19, 5 (2004), 140-153.
- [58] Leffingwell, D. and Widrig, D. *Managing Software Requirements: A Use Case Approach* (2 ed.). Pearson Education (2003).
- [59] Loeliger, J. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development* (1st ed.). O'Reilly Media (2009).
- [60] Malone, T. W. and Crowston, K. The interdisciplinary study of coordination. *ACM Comput. Surv.* 26, 1 (1994), 87-119.
- [61] Maraia, V. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional (2005).
- [62] McConnell, S. *Rapid Development: Taming Wild Software Schedules* (1st ed.).

Microsoft Press (1996).

- [63] McDowell, C., Werner, L., Bullock, H., and Fernald, J. The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE Bull.* 34, 1 (2002), 38-42.
- [64] McDowell, C., Werner, L., Bullock, H. E., and Fernald, J. The impact of pair programming on student performance, perception and persistence. In *Proc. of Intl. Conf. on Software Engineering* (2003), 602-607.
- [65] McDowell, C., Werner, L., Bullock, H. E., and Fernald, J. Pair programming improves student retention, confidence, and program quality. *Commun. ACM* 49, 8 (2006), 90-95.
- [66] McGuffin, L. and Olson, G. ShrEdit: a shared electronic workspace. *CSMIL Technical Report #45* (University of Michigan, 1992).
- [67] Mens, T. A State-of-the-Art Survey on Software Merging. *IEEE Trans. Softw. Eng.* 28, 5 (2002), 449-462.
- [68] Microsoft. *A history of Windows*.
<http://windows.microsoft.com/en-US/windows/history>
- [69] Müller, M. M. and Tichy, W. F. Case study: extreme programming in a university environment. In *Proc. of Intl. Conf. on Software Engineering* (2001), 537-544.
- [70] Munson, J. and Dewan, P. A concurrency control framework for collaborative systems. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (1996), 278-287.
- [71] Newman-Wolfe, R. E. and Pelimuhandiram, H. K. MACE: a fine grained concurrent editor. In *Proc. of Conf. on Organizational Computing Systems* (1991), 240-254.
- [72] Newman-Wolfe, R. E., Webb, M. L., and Montes, M. Implicit locking in the ensemble concurrent object-oriented graphics editor. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (1992), 265-272.

- [73] Nosek, J. T. The case for collaborative programming. *Commun. ACM* 41, 3 (1998), 105-108.
- [74] Perry, D., Siy, H., and Votta, L. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.* 10, 3 (2001), 308-337.
- [75] Pilato, C., Collins-Sussman, B., and Fitzpatrick, B. *Version Control with Subversion (2 ed.)*. O'Reilly Media (2008).
- [76] Pressman, R. *Software Engineering: A Practitioner's Approach (7 ed.)*. McGraw-Hill (2009).
- [77] Raynal, M. and Singhal, M. Logical Time: Capturing Causality in Distributed Systems. *Computer* 29, 2 (1996), 49-56.
- [78] Reeves, M. and Zhu, J. Moomba - a collaborative environment for supporting distributed extreme programming in global software development. *Extreme Programming and Agile Processes, LNCS*, Springer, 3092 (2004), 38-50.
- [79] Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A. Virtual Network Computing. *IEEE Internet Computing* 2, 1 (1998), 33-38.
- [80] Salinger, S., Oezbek, C., Beecher, K., and Schenk, J. Saros: an eclipse plug-in for distributed party programming. In *Proc. of ICSE Workshop on Cooperative and Human Aspects of Softw. Eng.* (2010), 48-55.
- [81] Sarma, A. A Survey of Collaborative Tools in Software Development. *ISR Technical Report #UCI-ISR-05-3* (Institute for Software Research, University of California, Irvine, 2005).
- [82] Schümmer, T. and Haake, J. M. Supporting distributed software development by modes of collaboration. In *Proc. of European Conf. on Computer Supported Cooperative Work* (2001), 79-98.
- [83] Schümmer, T. and Lukosch, S. Understanding Tools and Practices for Distributed Pair Programming. *Journal of Universal Computer Science* 15, 16 (2009),

3101-3125.

- [84] Sengupta, B., Chandra, S., and Sinha, V. A research agenda for distributed software development. In *Proc. of Intl. Conf. on Softw. Eng.* (2006), 731-740.
- [85] Shen, H. Internet-based Collaborative Programming Techniques and Environments. *PhD Thesis* (Griffith University, 2003).
- [86] Shen, H. and Sun., C. Flexible notification for collaborative systems. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2002), 77-86.
- [87] Shen, H. and Sun, C. Recipe: a web-based environment for supporting real-time collaborative programming. In *Proc. of Intl. Conf. on Networks, Parallel and Distributed Processing* (2002), 283-288.
- [88] Sommerville, I. *Software Engineering (9th Ed.)*. Addison Wesley (2010).
- [89] Stefik, M., Bobrow, D. G., Foster, G., Lanning, S., and Tatar, D. WYSIWIS revised: early experiences with multiuser interfaces. *ACM Trans. Inf. Syst.* 5, 2 (1987), 147-167.
- [90] Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. The structure and value of modularity in software design. In *Proc. of European Software Engineering Conf. held jointly with ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering* (2001), 99-108.
- [91] Summers, B. and Summers, R. *Official Microsoft Netmeeting 2.1 Book (1st ed.)*. Microsoft Press (1998).
- [92] Sun, C. Optional and responsive fine-grain locking in Internet-based collaborative systems. *IEEE Trans. Parallel Distrib. Syst.* 13, 9 (2002), 994-1008.
- [93] Sun, C. *OT FAQ: Operational Transformation Frequently Asked Questions and Answers*. <http://cooffice.ntu.edu.sg/otfaq>
- [94] Sun, C. and Ellis, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of ACM Conf. on Computer Supported*

Cooperative Work (1998), 59-68.

- [95] Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.* 5, 1 (1998), 63-108.
- [96] Sun, C. and Sasic, R. Optimal locking integrated with operational transformation in distributed real-time group editors. In *Proc. of Annual ACM Symposium on Principles of Distributed Computing* (1999), 43-52.
- [97] Sun, C., Wen, H., and Fan, H. Operational transformation for orthogonal conflict resolution in real-time collaborative 2d editing systems. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2012), 1391-1400.
- [98] Sun, C., Xia, S., Sun, D., Chen, D., Shen, H., and Cai, W. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.* 13, 4 (2006), 531-582.
- [99] Sun, D. and Sun, C. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Trans. Parallel Distrib. Syst.* 20, 10 (2009), 1454-1470.
- [100] Sun, D., Sun, C., Xia, S., and Shen, H. Creative conflict resolution in realtime collaborative editing systems. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2012), 1411-1420.
- [101] Tichy, W. F. RCS—a system for version control. *Softw. Pract. Exper.* 15, 7 (1985), 637-654.
- [102] Tichy, W. F. Tools for software configuration management. In *Proc. of the Intl. Workshop on Software Version and Configuration Control* (1988), 1-20.
- [103] Van Genuchten, M. Why is Software Late? An Empirical Study of Reasons for Delay in Software Development. *IEEE Trans. Softw. Eng.* 17, 6 (1991), 582-590.
- [104] Vessey, I. and Sravanapudi, A. P. CASE tools as collaborative support technologies. *Commun. ACM* 38, 1 (1995), 83-95.

- [105] Whitehead, J. Collaboration in software engineering: a roadmap. In *Future of Softw. Eng.* (2007), 214-225.
- [106] Wikipedia. *Development of Windows XP*.
http://en.wikipedia.org/wiki/Development_of_Windows_XP
- [107] Williams, L. A. and Kessler, R. R. All I really need to know about pair programming I learned in kindergarten. *Commun. ACM* 43, 5 (2000), 108-114.
- [108] Williams, L. A. and Kessler, R. R. The effects of “pair-pressure” and “pair-learning” on software engineering education. In *Proc. of Conf. on Software Engineering Education & Training* (2000), 59- 65.
- [109] Williams, L., Kessler, R. R., Cunningham, W., and Jeffries, R. Strengthening the case for pair programming. *IEEE Softw.* 17, 4 (2000), 19-25.
- [110] Williams, L. and Upchurch, R. L. In support of student pair-programming. *ACM SIGCSE Bull.* 33, 1 (2001), 327-331.
- [111] Xia, S., Sun, D., Sun, C., Chen, D., and Shen, H. Leveraging single-user applications for multi-user collaboration: the cword approach. In *Proc. of ACM Conf. on Computer Supported Cooperative Work* (2004), 162-171.

Publications Derived from this Research

- [1] **Hongfei Fan**, Chengzheng Sun, and Haifeng Shen. ATCoPE: any-time collaborative programming environment for seamless integration of real-time and non-real-time teamwork in software development. In *Proceedings of ACM International Conference on Supporting Group Work (GROUP 2012)*, 107-116.
- [2] **Hongfei Fan** and Chengzheng Sun. Supporting semantic conflict prevention in real-time collaborative programming environments. *ACM SIGAPP Applied Computing Review* 12, 2 (2012), 39-52.
- [3] **Hongfei Fan** and Chengzheng Sun. Achieving integrated consistency maintenance and awareness in real-time collaborative programming environments: the CoEclipse approach. In *Proceedings of IEEE International Conference on Computer Supported Cooperative Work in Design (CSCWD 2012)*, 94-101. **{Best Paper Award}**
- [4] **Hongfei Fan** and Chengzheng Sun. Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming. In *Proceedings of ACM Symposium on Applied Computing (SAC 2012)*, 737-742. **{Best Paper Award}**