CONSISTENCY MAINTENANCE IN COLLABORATIVE
GRAPHICS EDITING SYSTEMS

By
David Chen

A DISSERTATION SUBMITTED IN FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
THE SCHOOL OF COMPUTING AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING AND INFORMATION TECHNOLOGY
GRIFFITH UNIVERSITY
BRISBANE, QLD 4111, AUSTRALIA
NOVEMBER 2001

© Copyright by David Chen, 2001
Date: November 2001

Author: David Chen

Title: Consistency Maintenance in Collaborative Graphics Editing Systems

Department: School of Computing and Information Technology

Degree: Ph.D. Year: 2001

I declare this work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Signature of Author

THE AUTHOR RESERVES OTHER PUBLICATION RIGHTS, AND NEITHER THE THESIS NOR EXTENSIVE EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT THE AUTHOR’S WRITTEN PERMISSION.
# Table of Contents

Table of Contents iii

List of Figures vi

Acknowledgements viii

Publications derived from this research ix

Abstract xii

1 Introduction 1
   1.1 CSCW/Groupware .......................... 1
       1.1.1 Classification .......................... 2
       1.1.2 Applications .......................... 3
       1.1.3 Design issues .......................... 8
   1.2 Collaborative editing systems ................ 11
       1.2.1 Collaborative graphics editing systems ...... 13
       1.2.2 System architecture .................... 15
   1.3 Consistency maintenance ..................... 20
       1.3.1 Related work .......................... 20
       1.3.2 Collaborative graphics editing systems ...... 25
   1.4 Summary of contributions .................. 41
   1.5 Thesis outline ............................ 43

2 Maintaining consistent execution effect by conflict resolution 44
   2.1 Operation conflicts and multiple versions ........ 44
       2.1.1 Conflict and compatible relations .......... 44
       2.1.2 Accommodating all operation effects ........ 46
       2.1.3 Combined effect rules .......................... 50
2.2 Combined effects for any group of operations .......................... 54
  2.2.1 Conflict relation matrix and triangle ................................ 54
  2.2.2 Compatible groups set ................................................. 55
  2.2.3 Equivalent CGS .......................................................... 57
  2.2.4 Normalized CGS ......................................................... 58
  2.2.5 Combined effect specified by NCGS ................................. 61

2.3 Incremental creation of multiple versions ................................. 63
  2.3.1 The MOVIC algorithm ................................................. 64
  2.3.2 Order independence property ....................................... 67

2.4 Consistent object identification ............................................ 70
  2.4.1 Requirements for object identification .............................. 70
  2.4.2 Analysis of object identification issues ............................ 71
  2.4.3 The COID scheme ...................................................... 74

2.5 Revised conflict definition .................................................. 78

2.6 Version creation .............................................................. 84

2.7 Convergence ................................................................. 85
  2.7.1 Intra-object convergence ............................................. 85
  2.7.2 Inter-object convergence ............................................. 86

2.8 Comparison within the graphics editing domain .......................... 88

2.9 Comparison between text and graphics domain .......................... 91
  2.9.1 Comparison ............................................................. 91
  2.9.2 Generalization ........................................................ 96

2.10 Conclusions ........................................................................ 99

3 Maintaining consistent undo effect of any operation .................... 102
  3.1 Any Undo ....................................................................... 102

3.2 Generic Any Undo effect ...................................................... 104

3.3 Issues in graphics DO, UNDO, and REDO ................................. 105

3.4 Undo effect on objects ........................................................ 108
  3.4.1 Undo/redo effect .......................................................... 108
  3.4.2 Undo/redo effect generation .......................................... 116
  3.4.3 Undo operation representation ....................................... 117

3.5 Undo effect on multiple versions ............................................. 118
  3.5.1 Visible undo/redo effect on multiple versions ..................... 119
  3.5.2 Multiple version effect ................................................ 122
  3.5.3 Version removal ........................................................ 126
  3.5.4 Version re-creation .................................................... 129

3.6 Object identification .......................................................... 134
3.7 Undo and redo algorithms ........................................ 136
3.8 Operation specific issues in undo/redo ......................... 137
3.9 Comparisons .................................................. 139
3.10 Conclusion .................................................. 141

4 Maintaining consistency by conflict prevention ............... 143
  4.1 Locking .................................................. 143
  4.2 Locking in GRACE ........................................ 146
    4.2.1 To lock or not to lock? ............................... 146
    4.2.2 Responsive operation generation .................... 147
    4.2.3 Inconsistency problems .............................. 149
  4.3 Instant locking scheme .................................... 150
    4.3.1 Instant lock sharing ................................ 153
    4.3.2 Instant locking and concurrent versions .......... 155
  4.4 When does a lock stabilize? ............................... 157
  4.5 Conclusion ................................................ 158

5 Implementation issues ......................................... 161
  5.1 The site server process .................................. 161
  5.2 Operation generation and execution ....................... 162
  5.3 Any Undo implementation ................................ 165
  5.4 Instant locking implementation .......................... 166
  5.5 Maintaining executed operations .......................... 170
  5.6 Artificial network delay .................................. 172
  5.7 Usage experience ......................................... 174

6 Conclusion .................................................. 175
  6.1 Summary ................................................ 175
  6.2 Future work .............................................. 178

Bibliography .................................................. 181
List of Figures

1.1 Groupware time space matrix ........................................... 3
1.2 A centralized architecture, with single server process, for collaborative editing systems ........................................... 17
1.3 A replicated architecture for collaborative editing systems .......... 19
1.4 Concurrent transactions resulting in lost update ......................... 22
1.5 Intra-object divergence and intention violation problems caused by conflicting operations ........................................... 26
1.6 Inter-object divergence problem ........................................... 27
1.7 Out of causal order arrival ........................................... 28

2.1 The null-effect for conflicting operations .................................. 47
2.2 Single operation effect for conflicting operations ......................... 48
2.3 All operations effect for conflicting operations ........................... 49
2.4 A scenario for motivating CER3 ........................................... 52
2.5 CRM versus CRT ........................................... 55
2.6 A CRT and its corresponding CGS ........................................... 57
2.7 A CRT, and its corresponding CGS and NCGS ............................ 60
2.8 The CRT for Example 2.1 ........................................... 67
2.9 The CRT for Example 2.2 ........................................... 69
2.10 Operations’ generation order for Example 2.3 ............................ 77
2.11 The CRT for Example 2.3 ........................................... 78
2.12 The CRT for Example 2.4 ........................................... 81
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.13</td>
<td>The CRT with revised conflict definition for Example 2.3</td>
<td>84</td>
</tr>
<tr>
<td>2.14</td>
<td>The CRT with revised conflict definition for Example 2.4</td>
<td>84</td>
</tr>
<tr>
<td>2.15</td>
<td>The comparison of concurrency control schemes in REDUCE and GRACE</td>
<td>92</td>
</tr>
<tr>
<td>2.16</td>
<td>The intentions of Create operations are preserved, but divergence still occurs</td>
<td>94</td>
</tr>
<tr>
<td>3.1</td>
<td>The time-space diagram for Example 3.2</td>
<td>120</td>
</tr>
<tr>
<td>3.2</td>
<td>The CRT for Example 3.2</td>
<td>120</td>
</tr>
<tr>
<td>4.1</td>
<td>The operation generation process</td>
<td>149</td>
</tr>
<tr>
<td>4.2</td>
<td>Independent operations $O_1$ and $O_2$ may cause inconsistency problems</td>
<td>150</td>
</tr>
<tr>
<td>4.3</td>
<td>Region locks $R_1$ and $R_2$ with overlapping area $P$</td>
<td>153</td>
</tr>
<tr>
<td>4.4</td>
<td>Independent object and region locks where the target object $G$ is inside the region $R$</td>
<td>154</td>
</tr>
<tr>
<td>5.1</td>
<td>The internal structure of a Site Server process</td>
<td>163</td>
</tr>
<tr>
<td>5.2</td>
<td>Multiple versions of the rectangle are created</td>
<td>164</td>
</tr>
<tr>
<td>5.3</td>
<td>The undo table which lists all operations available for undo/redo</td>
<td>166</td>
</tr>
<tr>
<td>5.4</td>
<td>Different color outlines to represent different locking status</td>
<td>168</td>
</tr>
<tr>
<td>5.5</td>
<td>A locked object and region on site 1</td>
<td>169</td>
</tr>
<tr>
<td>5.6</td>
<td>A locked object and region on site 2</td>
<td>170</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to express my most sincere gratitude to my supervisor, Professor Chengzheng Sun, who I have known since when I was an undergraduate student. He has looked after me throughout my years of postgraduate study. He has taught me every aspect on how to be successful in academia. His pursuit for research excellence has set a good example for all his PhD students. I do not think there is anything more he could have done for me.

I would like to thank my associated supervisor, Professor Rodney Topor. His database classes during my undergraduate and honours years have stimulated my interest research. His critical comments have helped me in producing results of higher quality.

I am indebted to my parents. They have patiently supported me throughout my years of studies. Without their encouragement and support, this would not be possible. I am lucky to have such wonderful parents.

I would like to thank my friends and colleagues, Richard Wong, Fred Liang, Raymond Woo, Jerome Chik, and Sven Venema for their companionship during these years. The practical knowledge I have gained from our discussions, arguments, and various activities, is as valuable as what I have learnt from my research. Most importantly, we had a lot of fun along the way.

Last but not the least, I am indebted to the girl who I love with all my heart. She has provided me with emotional support which has carried me through some of the most difficult periods. Furthermore, she has given me the strength I needed to cross the finish line. I promised her that I will finish my thesis. Now I have done it!
Publications derived from this research


Abstract

Real-time collaborative editing systems are groupware systems which allow multiple users to edit the same document at the same time. Collaborative graphics editing systems can be used for CAD, CASE tools to draw design diagrams, or to draw illustrative figures within documents collaboratively. In these types of systems, operations may be generated concurrently which may result in conflicts. The execution of conflicting operations may produce inconsistent results. Consistency maintenance is one of the most significant challenges in designing and implementing real-time collaborative editing systems. This dissertation focuses on maintaining consistency in collaborative graphics editing systems which provide fast response time in the Internet environment and allow unconstrained editing.

In this thesis, a multiple object version scheme is first developed to solve the inconsistency problems caused by the execution of conflicting operations. This scheme preserves the work concurrently produced by multiple users in the face of conflict, and minimizing the number of object versions for conflict resolution. From this result, the users can make an informed decision on which operation’s effect they wish to keep. Major technical contributions of this work include a formal specification of a unique combined effect for an arbitrary group of conflict and compatible operations, a distributed algorithm for incremental creation of multiple object versions, a consistent object identification scheme for multiple object versions and a convergent layering scheme for overlapping objects. This multiple version scheme is the foundation upon which further research is built.

Secondly, an Any Undo scheme is developed to maintain consistency under the
condition that users may select any operation to undo at any time. Any Undo is especially important in collaborative editing systems because it can be used to support local or global undo and also multiple undo models including single-step, chronological, and selective undo. Major technical contributions include a generic specification of undo/redo effect of any graphics operation, algorithms to produce the required undo effect, and algorithms to maintain consistency by removing and recreating object versions. This is the only undo scheme which supports the concurrency control technique of the multiple object version. Furthermore, as far as is known, no other undo scheme in collaborative graphics editing systems has the flexibility of this Any Undo scheme.

Finally, an optional locking scheme is proposed which can be used to prevent the generation of conflicting operations while the responsiveness of the system is not sacrificed. This locking scheme is optional because, even if locks are not placed, syntactic consistency of the system is still maintained by the multiple object version scheme. Hence, the multiple object version scheme and the optional instant locking scheme are complimentary techniques designed to work together collaboratively. Two types of locks are proposed, object and region locks. Object locks are used to lock objects and their usefulness is obvious. Region locks are used to lock editing areas. In addition to conflict prevention, regions locks can be used for access control or as a group awareness mechanism.

Both theoretical and experimental methods have been applied during the development of the major concepts and algorithms in this research program. Formal definitions and proofs have been given to some major high level concepts and algorithms in the multi-version-based consistency maintenance approach. Furthermore, all algorithms and schemes proposed in this thesis have been implemented in a prototype collaborative editing system called GRACE. GRACE has not only served as a vehicle to motivate, test, and demonstrate the new ideas and techniques during this research program, and will also be used as a vehicle for usability study in the future.
Chapter 1

Introduction

1.1 CSCW/Groupware

Starting from late in the 20th century, there has been an unprecedented growth of the computer and communication industries. The rapidly growing popularity of the Internet has accelerated the integration of computers and communication networks. Instead of creating applications that allow a person to pursue only individual work, many researchers and developers began thinking about systems that allowed people to work together [35]. A new paradigm of human to human interaction via computer was created and the discipline of ‘Computer Supported Cooperative Work’ (CSCW) [2, 39, 44] arose.

“CSCW is a scientific discipline that guides the thoughtful and appropriate design and development of computer systems to support group work [39].”

Despite the name, this field of study is not restricted to issues of ‘cooperation’ or ‘work’ but also examines competition, socialization, and play [8]. The multi-user software supporting CSCW systems are known as groupware. Ellis, Gibbs and Rein
defined groupware as:

“computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment [32].”

Groupware offers significant advantages over single-user systems. Some of the most common reasons people want to use groupware are to [8]:

- facilitate communication: make it faster, clearer, more persuasive;
- enable communication where it would not otherwise be possible;
- enable telecommuting: cut down on travel costs;
- bring together multiple perspectives and expertise;
- form groups with common interests where it would not be possible to gather a sufficient number of people face-to-face;
- save time and cost in coordinating group work;
- facilitate group problem-solving; and
- enable new modes of communication, such as anonymous interchanges or structured interactions.

1.1.1 Classification

The time-space matrix in Figure 1.1 is often used to classify groupware [32]. Groupware can be used to support asynchronous or synchronous interaction. Asynchronous groupware, such as electronic mail or bulletin board, transcends the limitations of
time, allowing communication and decision making among groups of individuals at different times. In contrast, synchronous groupware assists a group of individuals in working together at the same time to carry out tasks such as communicating, making decisions, planning new initiatives, structuring proposals, writing papers, or sketching designs [2].

The other dimension of classification regards where the participants are located. Groupware can be conceived to help a face-to-face (co-located) group, or a group that is distributed over many locations. For example, the presentation support feature of electronic meeting rooms require users to be in the same place, where as computer conferencing systems support users in different locations.

<table>
<thead>
<tr>
<th>Same Place</th>
<th>Same Time</th>
<th>Different Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>co-located</td>
<td>Voting systems, presentation support</td>
<td>Shared computers</td>
</tr>
<tr>
<td>Different Place</td>
<td>Videophones, Chat systems</td>
<td>Email, Newsgroups</td>
</tr>
</tbody>
</table>

Figure 1.1: Groupware time space matrix

1.1.2 Applications

Groupware includes a wide spectrum of systems. These systems can be classified into several categories according to their application level functionalities [32] (some groupware may belong to more than one category):

- **Multiuser editors.** Multiuser editors, or collaborative editing systems, allow members of a team to work on the same document concurrently. Collaborative editing systems can be either synchronous or asynchronous. Word processors
may provide asynchronous support by showing authorship and by allowing users
to track changes and make annotations to documents. Authors collaborating on
a document may also be given tools to help plan and coordinate the authoring
process, such as methods for locking parts of the document linking separately-
authored documents. Synchronous support allows authors to see each other’s
changes instantly. These types of systems will be discussed extensively in the
rest of this dissertation.

- **Message systems.** These types of systems support the exchange of textual
  messages between groups of users. They are by far the most widely used group-
  ware. There are several variations of message systems.

  - *Email* is the most well known message system in use today. While the basic
technology is designed to pass simple messages between two people, even
  relatively basic email systems today typically include interesting features
  for forwarding messages, filing messages, and creating mailing groups.

  - *Newsgroups* are similar in concept to email systems except that they are
    intended for messages among large groups of people instead of one-to-one
    communication. To communicate, messages are left in newsgroups. Any
    interested reader can go to that newsgroup and retrieve the messages. In
    comparison to email systems, newsgroups only show messages to a user
    when this user explicitly requests these messages (an ‘on-demand’ ser-
    vice), while email systems deliver messages as they become available (an
    ‘interrupt-driven’ interface).
- Instant messaging programs have recently become popular. Instant messaging programs provide near-synchronous one-to-one message communication [72]. Some examples of such programs are ICQ, AOL's Instant Messenger, and Microsoft Messenger Service.

- Chat systems permit people to write messages in real-time to each other. As each person submits a message, it appears at the bottom of the a scrolling screen. Most instant messaging programs also provide support (although limited) for real-time chat where private chat sessions can be created for a small group of people. Other chat systems, such as Internet Relay Chat (IRC) supports large chat groups with more sophisticated features. These systems allow the creation of chat rooms (public or private), with controlled access or with moderators to lead the discussions.

- Group decision support systems and electronic meeting rooms.
  Group decision support systems (GDSSs) are used to improve the productivity of decision-making meetings, either by speeding up the decision-making process or by improving the quality of the resulting decisions. They provide tools for brainstorming, critiquing ideas, putting weights and probabilities on events and alternatives, and voting. Such systems enable presumably more rational and even-handed decisions. Primarily designed to facilitate meetings, they encourage equal participation by, for instance, providing anonymity or enforcing turn-taking. Often GDSSs are implemented as electronic meeting rooms that contain several networked workstations, large computer-controlled public displays, and audio/video equipment. Examples of such systems are Colab [96], Cognoter [111], Tivoli [71], We-Met [116], Helsinki [64], TeamRooms [92], and
DOLPHIN [98, 65].

- **Computer conferencing.** Computer conferencing allows a group of users, who are either gathered in an electronic meeting room or physically dispersed, to interact synchronously through their workstations or terminals. When a group is physically dispersed, an audio and video link is often established. Video is advantageous when visual information is being discussed, but may not provide substantial benefit in most cases where conventional audio telephones are adequate. In addition to supporting conversations, video may also be used in less direct situations, such as providing a view of activities at a remote location. Examples of computer conferencing systems are SMART 2000 [66], MAJIC [79], Meme Tags and Community Mirror [7], and HS113 [97].

- **Intelligent agents.** Intelligent agents are responsible for a specific set of tasks, and the user interface makes their actions resemble those of other users. These agents are often used as additional players in multiplayer computer games [82] when the number of human participants is low for a challenging game. Another task of intelligent agents is to provide useful information to the users without being asked precise questions or even without explicit question at all [114].

- **Coordination systems.** Coordination systems address the problem of “integration and harmonious adjustment of individual work efforts towards the accomplishment of a larger goal” [95]. *Group calendars* is a type of coordination system. Group calendars allow scheduling, project management, and coordination among many people. Typical features of such a system detect when schedules conflict, or find meeting times that are suitable for everyone.
Group calendars also help to locate people.

- **Workflow systems.** Workflow systems [25, 12, 38] allow documents to be routed through organizations through a relatively fixed process. A simple example of a workflow application is an expense report in an organization: an employee enters an expense report and submits it, a copy is archived then routed to the employee’s manager for approval, the manager receives the document, electronically approves it and sends it on and the expense is registered to the group’s account and forwarded to the account department for payment [8]. Workflow systems may provide features such as routing, development of forms, and support for differing roles and privileges.

- **Multi-player games.** In the old days (late 1980s), only asynchronous and text-based multi-player games were possible. Nowadays, highly interactive real-time multi-player games with graphical user interface are common. They can be played at home, on people’s own personal computers or game consoles, connected via Local Area Networks (LAN) or the Internet. Some games allow users to create their own gaming sessions (running on their own computer). This type of session usually allows for a small group of players (normally less than ten). Other games provide dedicated game servers where users have to connected to the game servers to play (they also have to pay a monthly subscription fee). These types of sessions allow a large number of players (hundreds or even thousands) to participate in the same session.

One of the earliest commercial groupware that deserves a special mention is Lotus Notes [86]. The reason Notes was not classified above is because Notes is a unique
hybrid, including elements of messaging, group discussion, calendaring/scheduling, database management, forms and workflow systems. The tight integration of these functions, along with the control that Lotus handed over to the independent developer, made Notes uniquely suited to a vast array of business applications. Notes (and Domino, its Internet-age successor) went on to claim more than fifty million users.

1.1.3 Design issues

CSCW is a multi-discipline field which requires expertise from both social science and computer science [46]. Many expensive groupware failures are the result of not understanding the unique social dynamics of group work [47]. The study of social science and psychology can give us insights on how groups of people work together. This is important in deriving the user requirements for groupware. Having the knowledge of social science does not guarantee successful development of groupware. Groupware development is complex and requires expertise in many areas of computer science, such as human-computer interaction, networking and communication, operating systems, database systems, audio and video technology, and artificial intelligence.

Groupware systems introduce design problems not present in single-user systems. Some of the challenging issues faced by the designers of groupware are [8, 32]:

- **Group interfaces.** Group interfaces differ from single-user interfaces in that they depict group activity and are controlled by multiple users rather than a single user. A basic problem is how to manage complexity: multiple users can produce a higher level of activity and a greater degree of concurrency than single users, and the interface must support this complex behaviour. Other important questions are: What single-user interface techniques and concepts are useful for
constructing group interfaces? Where do they fail, pointing to the need for new concepts?

- **Awareness information.** Being aware of other users’ locations, activities, and intentions relative to the task enables people to work together more effectively. Awareness information takes many forms. In video conferencing, simply providing a wide-angle camera lens can provide a greater degree of environmental awareness. In email, simple information such as the time and date of the message or the signature file of the sender (i.e. with contact and company information, etc.) gives the context for making sense of the message. Awareness tools can be designed for letting others know when you are in the office or not, letting them know what document you are working on, or how you are feeling at a given time. The designers should also be aware that the over-provision of awareness information may violate the privacy of the the users.

- **Organization structure modeling.** Most actions in an organization have a known range of responses and personnel to handle them. When someone fills out an official form, that form usually has a pre-determined route that it takes through an organization; possibly to a manager for a signature, then an administrator for processing and filing, then perhaps a duplicate is sent back to the original employee. When the type of structure is known, systems can take advantage of the structure to speed up communications and minimize errors. However, exceptions are frequent since organizations are open systems [51]; in particular, they contain incomplete and partial information about their day-to-day activities, making it impossible to identify all the situations encountered.
• **Session control.** A session is a situation where a group of people are in a conversation together at a given time, such as a group of people together in a chat room, working on the same document, or playing the same instance of a game. Metaphorically, session control is like a person standing at the door of a room checking IDs and deciding who gets to go in. Session control issues include finding out what rooms are available, determining who can enter and exit the room, and when and how.

• **Access control.** Once people have joined a session, it must be decided what kind of access each person has to the shared resources. For instance, when using a shared whiteboard, can everyone draw on it at the same time (simultaneous access), can only one person access it at a time (by passing a token or baton), is there a moderator who controls access, and is there a time limit for each person? Simultaneous access by everyone to everything is often the preferred method for the most fluid session. However, it is also useful to provide some kind of mediated access to prevent unauthorized access and to avoiding people making conflicting changes. Access control should be adjustable to fit different group processes.

• **Concurrency control.** Groupware systems needs concurrency control (or consistency maintenance) schemes to resolve inconsistency problems caused by participants’ simultaneous operations. While traditional systems seek to provide the illusion that the users are isolated, groupware makes the user aware that s/he is part of a group. Therefore, many concurrency control techniques which worked well in traditional systems have been found to be no longer appropriate in groupware [32, 42, 62]. For example, many of the approaches to handling
concurrency in database applications, such as explicit locking or transaction processing, are not only inappropriate for groupware but can actually hinder tightly coupled teamwork. Therefore, new concurrency control techniques need to be developed especially for groupware.

1.2 Collaborative editing systems

A particular type of groupware is the collaborative editing system. This type of system allows physically dispersed members of a group to jointly compose, view, and edit a document (or ‘object’ to be more general). The types of documents that can be edit collaboratively include text, bitmap graphics, object graphics, spreadsheet, sound, and video, etc. Collaborative editing systems can be used specifically for:

- collaboration in: writing/authoring [110, 74, 56], learning [75], designing [93] of hardware (computers, buildings, cars etc.) or software, programming, reviewing, and debugging;

- concurrent data entrying to the same document (i.e. entering results into a table);

- recording ideas during brainstorming sessions [52]; and

- holding a meeting [80, 83, 116] or conference [91, 66].

As with all groupware systems, collaborative editing systems can also be asynchronous or synchronous. With asynchronous collaborative editing systems, documents are stored in a central repository/database. Before a document can be edited, a copy is made from a central repository to the local site. Users edit their own local
copy independent of other users. After some editing, update is triggered specifically to update the document or save it as a different version in the repository. Examples of such systems are CVS [87], Instant Update [110], PREP [74], and Quilt [60]. For this type of system, simultaneous changes within a single file, or across a whole collection of files, will logically conflict with one another. The most common solution is to indicate the occurrence of conflict and let the users resolve it [73, 87].

Synchronous or real-time collaborative editing systems allow multiple users to edit the same document at the same time from different sites. These systems are highly interactive, where changes made to the document are automatically updated to all sites instantly. There are many different varieties of synchronous collaborative editing systems, distinguished by their characteristics. In some systems, the response time is dependent on the latency of the underlying network, whereas other systems provide fast response time independent of the network. Some systems are designed only to work in low latency Local Area Networks (LAN). These types of systems would not work very well in Wide Area Networks (WAN) with higher and non-deterministic latency. Another characteristic is the editing restriction placed by the system. Some systems only allow input from one site at a time, whereas other systems allow concurrent inputs on different sections/objects. Furthermore, other systems allow users the freedom to edit any part of the document at any time.

This dissertation focuses on real-time collaborative graphics editing systems. The aim of this research is to investigate and develop innovative technologies for the construction of collaborative graphics editing systems which satisfy the following requirements [105]:

1. Real-time: the response to local user actions is quick (ideally, as quick as a
single-user editor), and the latency for reflecting remote user actions is low (determined by the external communication latency only).

2. Distributed: collaborating users may operate different machines connected via different communication networks with nondeterministic latency.

3. Unconstrained: multiple users are allowed to concurrently and freely edit any part of the document at any time, in order to facilitate free and natural information flow among multiple users, as advocated by Dourish [28], Ellis and Gibbs [31], Hymes and Olson [52], Koch [56], Ressel et al. [90], and Sun et al. [108].

In the next section, collaborative graphics editing systems are introduced to define the scope of this research. Then different architectures for collaborative editing systems are examined to determine which architecture is most suitable in satisfying the above three requirements.

1.2.1 Collaborative graphics editing systems

Graphics documents can be roughly classified into two types: object based, or bitmap based. A bitmap (also called ‘raster’) graphics document is created from rows of differently colored pixels that together form an image. In their simplest form, bitmaps have only two colors, with each pixel being either black or white. With increasing complexity, an image can include more colors; photograph-quality images may have millions.

Object (also known as ‘vector’) graphics documents are constructed using mathematical formulas describing objects (shapes), colors, and placement. Rather than a grid of pixels, an object graphics document consists of objects such as line, rectangle,
circle, etc., which together make a picture. While a bitmap image contains information about the color of each pixel, an object graphics document contains instructions about where to place each of the objects. It is even possible to embed a bitmap graphic within an object graphics document, which is how object-bitmap ‘hybrid’ graphics work. It is not possible, however, to embed object information within a bitmap.

Bitmap and object graphics both have their strengths and weaknesses. Hence, they are used for different purposes. Bitmap graphics editing systems are commonly used for photo editing, creating images, and as a whiteboard to scribble ideas. Object graphics editing systems are used for CAD, CASE tools to draw design diagrams, and to draw simple diagrams within documents or for presentations.

Due to the different natures of bitmap and object graphics collaborative editing systems, the design issues for these systems are also different. We intend to examine both bitmap and object graphics editing systems. However, due to their complexity, this research will focus on object graphics collaborative editing systems. The rationale for selecting object graphics over bitmap graphics is to enable integration of collaborated text and object graphics editing systems to produce a collaborated word processor. Bitmap graphics systems will be investigated subsequently. The object graphics collaborative editing system developed as part of this research is called GRACE (GRAphics Collaborative Editing system).

For this purpose, each graphical object will be regarded as a collection of attributes such as type, size, position, color, and group, etc. Operations in object graphics editing systems can be classified into three general categories:

- Operations to create objects. These operations are used to create and set the
initial attribute values of objects. There is only one type of operation in GRACE under this category, it is the Create operation (which can create different types of objects).

- Operations to remove objects. These operations are used to remove existing objects from the document. There is only one type of operation in GRACE under this category, it is the Destroy operation.

- Operations to update attribute values. These operations are used to update/change attribute values of existing objects. Currently, there are three types of operations in GRACE under this category, they are Move, Fill, and Resize. For example, a Move operation changes the position attribute value of the object it is applied to. It should be emphasized that update operations specify changes by the attribute's absolute value (instead of the relative value). For example, a Move operation specifies the position to move an object to (not the distance to move by). Even if the editor allows the user to generate an operation by specifying a relative value, this relative value can be converted into absolute value before the operation is executed.

In the next section, the different architecture of collaborative editing systems are examined to determine how the three requirements of real-time, distributed, and unconstrained editing can be satisfied for GRACE.

### 1.2.2 System architecture

The central issue regarding the architecture of systems with a shared document (or data) is where the document is stored and how this document is maintained. In
general, the storage of the shared document can be centralized, replicated or a hybrid of both approaches.

**Centralized architecture**

With this architecture, there is a distinction between the editing sites and the server site. There can be multiple editing sites, each running a client process. The client process is responsible for generating operations from user inputs to update the shared document and displaying the content of the shared document on the local user interface.

There is only one server site. The shared document is kept at the server site. The server process is responsible for updating the shared document and sending the content of user interface to the client for display. There are two strategies for updating the shared document.

1. There is only one server process (as shown in Figure 1.2). Since only the server process can access the document, there is no concurrent access to the shared document hence no inconsistency problem. However, not having concurrent access is also a disadvantage because it limits the system performance since, by definition, only one user can access the document at a time. If a user is issuing a lengthy operation (e.g. moving an object), no other user can access the shared document during this time. Systems use this architecture to provide the illusion of concurrent access by supporting only fine-grained operations which can be executed in a very short time (e.g. set the color of a pixel in a bitmap document). WSCRAWL [115] is a bitmap based collaborative graphics editing system with such an architecture.
Figure 1.2: A centralized architecture, with single server process, for collaborative editing systems

2. There are multiple server processes which can access the shared document concurrently. This is a common approach in database systems [33]. The increase in performance is achieved at the cost of possible inconsistencies caused by concurrent access of the shared document. These inconsistency problems need to be resolved by using concurrency control methods.

No matter whether there are single or multiple server processes, the major disadvantage for the centralized architecture is that the local response time may be long. This is due to the fact that once an operation is generated, its execution is reflected on the local user interface only after the following steps:
1. the operation is sent to the server;

2. the server executes the operation, then sends a message (containing user interface information) to inform all clients of the update; and

3. the client receives the reply message from the server and update the local user interface.

The speed of completion for these three steps is highly dependent on the network latency. In a network where the latency is high (e.g. the Internet), the response time is long. Since collaborative editing systems with centralized architecture cannot guarantee fast response time, this architecture does not satisfy the requirements of GRACE.

**Replicated architecture**

With the replicated architecture [31, 43, 53, 71, 105], there is no server site. The server process and the shared document are replicated at all editing sites. Hence, each editing site contains a client process, a server process, and a copy of the shared document. The server process at each site is responsible for maintaining the consistency of its copy of the shared document. The server processes communicate directly with each other to ensure all updates are available at all sites for execution. This architecture is shown in Figure 1.3.

Collaborative editing systems with replicated architecture can provide fast response time with optimistic execution. When a local operation is generated, it is executed immediately by the local server process and the result updated to its client instantly (without requiring network communication). Hence the response time is independent of the network latency. However, the disadvantage of this approach is
that inconsistencies on replicated copies may occur due to concurrent update to the shared document made by multiple server processes.

**Hybrid architecture**

In addition to centralized and replicated architecture, a hybrid of these two architectures is also possible. With the hybrid architecture [76], each editing site contains a client and a server process plus a copy of the shared document. In addition, there is also a server site containing the shared document and a server process.
When operations are generated, those which do not cause inconsistency when executed concurrently (i.e. Create operations) are allowed to be executed locally before being sent to remote sites for execution (including the server site). This will produce good response time. However, other types of operations (all update operations) need to be executed via the server site to avoid inconsistency. The response time for these types of operations is still dependent on the network latency.

Only replicated architecture can satisfy the three requirements of GRACE. Furthermore, inconsistency associated with this architecture can be prevented or resolved by applying concurrency control techniques. Therefore, a replicated architecture with optimistic execution is adopted for GRACE. The challenge in building GRACE is to design concurrency control techniques to maintain consistency.

1.3 Consistency maintenance

In the past, the research in consistency maintenance has been driven by the work in database and distributed systems. Therefore, consistency maintenance issues in these systems are first presented. Consistency maintenance issues in collaborative graphics editing systems are then examined.

1.3.1 Related work

This section will first discuss consistency maintenance in database systems. Then a specific type of distributed system called 'distributed shared memory' is presented for its rich set of consistency models.
Database systems

Large database systems allow concurrent access to the database (to improve efficiency). However, the execution of concurrent transactions may produce inconsistent result because these transactions may be interleaved in arbitrary orders. For example, there are two transactions. Transaction 1 (T1) consists of three operations:

1. $TMP1 := \text{read\_item}(X)$; which fetches a data item $X$ from the database into the variable (in memory) $TMP1$

2. $TMP1 := TMP1 - N$; which decreases $TMP1$ in memory by $N$

3. $\text{write\_item}(X, TMP1)$; which writes the value of $TMP1$ back to $X$

Transaction 2 (T2) also consists of three operations which update the same data item:

1. $TMP2 := \text{read\_item}(X)$

2. $TMP2 := TMP2 + M$

3. $\text{write\_item}(X, TMP2)$

The execution of these two transactions are interleaved so that the first two operations in T1 are executed first, followed by the first two operations in T2, then operation 3 in T1 followed by operation 3 in T2 (as shown in Figure 1.4). The end result of this execution is that the effect of T1 is lost because the value set by T1 is overwritten by T2. Imagine $X$ represents the number of seats reserved in a flight. If $X = 80$ at the start, $N = 5$ (T1 cancels five seats), and $M = 4$ (T2 reserves four seats), the final result should be $X = 79$. However, the execution of T1 and T2 resulted in $X = 84$ because the update that canceled five seats was lost.
Figure 1.4: Concurrent transactions resulting in lost update

The consistency property that database systems have to preserve is *serial equivalence* [6, 5]. This property requires that the effect produced for the execution of a set of transactions be equivalent to the effect produced when these transactions are executed one at a time in some order. This consistency property does not restrict the execution order of operations, as long as the serial equivalence effect is produced. In the above example, if T1 and T2 are executed one at a time in either order, the correct result of \( X = 79 \) will be produced.

This consistency property is used as a criterion for derivation of concurrency control schemes. The concurrency control schemes for maintaining serial equivalence include locking and timestamp ordering (these schemes will be discussed later).

In collaborative editing systems, if serial equivalence is preserved, then after the execution of the same set of operations, copies of the shared document will be identical at all sites. However, serial equivalence is too restrictive because it does not allow for optimistic execution in a replicated architecture. In collaborative editing systems, the effects of operations appear immediately after their execution. By using optimistic execution, concurrently generated operations may appear in different orders.
on different sites. Since optimistic execution is vital in providing fast response time, a consistency property is required that guarantees the copies of the same document are identical at all sites yet allowing optimistic execution.

**Distributed shared memory**

Distributed shared memory (DSM) systems are another class of distributed systems which provide the illusion of shared memory for a collection of computers even though these computers do not have physically shared memory. The replicated architecture is commonly used to increase performance. With this approach, pages of the shared memory are replicated to the local memory of each computer. If a replicated page is updated, it may become inconsistent with other replicas. Before answering the question of how to maintain replica consistency, one must know the consistency requirements.

Like database systems, memories are accessed by two operations (read and write) to a particular memory unit (a unit can be a variable, an object, or a page). In a physically shared memory system, a read operation returns the most recently written value. This model of consistency is called *strict consistency*. Achieving strict consistency is very inefficient in DSM. Furthermore, most applications using DSM do not require strict consistency. Therefore, consistency models with weaker consistency requirements have been proposed.

- *Sequential consistency* [58]. The result of any execution is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in this sequence in the order specified by its program.
• *Causal consistency* [1]. Writes that are causally related (refer to Definition 1.1) must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different computers.

• *PRAM consistency* [63]. Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

• *Weak consistency* [30]. The view of sequential consistency is provided if the program obeys the synchronization model by explicitly inserting synchronization variables at the required positions.

There are two general techniques for maintaining the consistency of the replica: write-update and write-invalidate. These are applicable to a variety of DSM consistency models. The outline of these techniques are:

• *Write-update:* The write operation is allowed to take place locally, but the value written is sent to the processes containing a replica. When a process receives the update message, it then updates the local copy with the new value. The memory consistency model that is implemented by write-update mainly depends on the quality of service provided by the underlying network. For example, sequential consistency can be achieved if updates are sent via totally ordered multicast.

• *Write-invalidate:* When a page is modified, an invalidation message is sent to the processes containing a replica. When this message is received by a process, its local copy is marked as invalid. The final result is that there will only be one copy left, so inconsistency problems are avoided. Subsequently, when a process
tries to access that part of the memory, a copy is made from other process containing it.

Different types of systems have different inconsistency problems and different consistency models, and require different concurrency control techniques. The background knowledge on consistency maintenance problems, requirements, and techniques, although not necessarily directly applicable, serves as the inspiration behind the development of many real-time collaborative editing systems. For instance, many collaborative editing systems adopt serial equivalence as their consistency model and use locking and timestamp ordering as the concurrency control techniques. Furthermore, techniques similar to write-update are used in many collaborative editing systems (including GRACE) for updating replica copies of the shared document.

1.3.2 Collaborative graphics editing systems

This section examines the consistency maintenance issues for collaborative graphics editing systems. The inconsistency problems are first presented. A consistency model is then defined. Finally, concurrency control techniques are examined to determine which problems have been solved (by which techniques) and which problems remain unsolved.

Inconsistency problems

For meeting the high responsiveness requirement, the optimistic operation execution on a replicated architecture is used. Hence, when an operation is generated, it is
executed on the replica of the shared document immediately, then broadcast to remote sites and executed there in its original form upon its arrival. There are three inconsistency problems which manifest themselves in collaborative graphics editing systems.

Divergence

Operations may arrive and be executed at different sites in different orders, resulting in different final results. The divergent results can be classified into two types: intra-object and inter-object. With intra-object divergence, different results are produced on the same object at different sites. For example, let operations $O_1$ and $O_2$ be $Move$ operations to move the same object, $G$, to two different positions, $X$ and $Y$ respectively. These operations are called conflicting operations. Conflicting operations may be executed in different orders at different sites (as shown in Figure 1.5). At Site 1,
$O_1$ is executed before $O_2$. So $G$ will be moved to $X$ by $O_1$, then to $Y$ by $O_2$. At Site 2, $O_2$ is executed before $O_1$. So $G$ will be moved to $Y$ by $O_2$, then to $X$ by $O_1$. The end result is that $G$ is at $Y$ in Site 1 and at $X$ in Site 2. Hence, intra-object divergence has occurred.

With *inter-object divergence*, different results are produced on two or more overlapping objects at different sites. For example, let $O_1$ and $O_2$ be *Create* operations to create two overlapping objects $G_1$ and $G_2$ respectively. Assuming a newly created object will be placed on top of any existing object. At Site 1, $G_2$ will be on top of $G_1$ since $O_2$ is executed last. At Site 2, $G_1$ will be on top of $G_2$ since $O_1$ is executed last. Hence, inter-object divergence has occurred as shown in Figure 1.6. Apparently, any divergent final result should be prohibited from collaborative editing systems where the consistency of the final results are required.
Figure 1.7: Out of causal order arrival

Causality violation

Due to nondeterministic communication latency, operations may arrive and be executed out of their natural cause-effect order. As shown in Figure 1.7, operation $O_2$ is generated after the arrival of $O_1$ at Site 2; the editing effect of $O_1$ on the shared document has been seen by the user at Site 2 at the time when $O_2$ is generated. Therefore, $O_2$ may be dependent on $O_1$ (see Section 1.3.2 for a more precise definition about dependency). However, since $O_2$ arrives and is executed before $O_1$ at Site 3, confusion may occur to the system as well as to the user at Site 3. For example, $O_1$ is a Create operation to create an object $G$ and $O_2$ is a Move operation to move $G$. If $O_2$ is executed before $O_1$, as happened in Site 3, then $O_2$ will be referring to an object that does not exist. Another example, suppose the users at Site 1 and Site 2 are playing a game of tic-tac-toe and the user at Site 3 is observing. User 1 at Site 1 started first by issuing its move with $O_1$. After User 2 at Site 2 has seen User 1’s move, s/he then makes his/her own move with $O_2$. However, User 3 at Site 3 will see User 2 move before User 1. This will cause confusion among the users about who
made the first move. Therefore, out-of-causal-order execution should be prohibited for the sake of system correctness and meeting the requirements of synchronized interaction among multiple users in many applications.

**Intention violation**

Conflicting operations may be generated to change the same attribute of the same object to different values. As a result, the effects of some operations are not preserved. For example, in Figure 1.5, \( O_1 \) and \( O_2 \) are *Move* operations to move the same object, \( G \), to different positions, \( X \) and \( Y \) respectively. Since \( O_1 \) and \( O_2 \) are moving the same object \( G \) to two different positions, it is impossible to accommodate their conflicting effects in the same target object. So either \( G \) will be moved to \( X \) or it will be moved to \( Y \), but not both. A consequence of this intention violation is that, whenever there is a conflict, only one user’s work can be preserved. Therefore, when a conflict occurs, users may not see a consistent and explicit picture of what other users have done or what their intentions were, and hence they may not be able to take proper actions to resolve their conflict [17, 56, 102]. Although the example used for this inconsistency problem is the same as in the divergence problem, divergence and intention violation are independent of each other. Even if convergence is somehow achieved (i.e. make \( G \) at the same position at all sites after execution), only the effect of one operation can be preserved and other conflicting operations’ effects are lost.

Apart from the *syntactic* inconsistency problems such as those identified above, there are still other *semantic* inconsistency problems, as pointed out by Zhang and Yang [119], Dourish [28, 29] and Sun et al. [108, 105]. For example, a diagram
contains the drawing of two baskets with the colors red and green respectively. The green basket is full of green apples. The red basket is full of red apples except for one, call it apple $A$, which is green. Two users concurrently generated operations to edit $A$. One user generated operation $O_1$ to move $A$ to the green basket. The other user generated operation $O_2$ to change the color of $A$ to red. Both users want to make the green basket contain only green apples and red basket contain only red apples. The execution of both operations would produce the result that $A$, with color red, is in the green basket. This result is syntactically correct since the effects of both $O_1$ and $O_2$ are preserved. However, it is semantically incorrect since the green basket now contains a red apple.

This kind of semantic inconsistency problem cannot be resolved by the underlying consistency maintenance mechanisms without the intervention of the users in collaboration and additional supporting mechanisms. As far as is known, none of the existing collaborative editing systems have attempted to maintain semantical consistency automatically. Therefore, the consistency maintenance approach to be presented in this thesis is not intended to address the semantic inconsistency problem either.

A consistency model

In this section a consistency model [108, 105] containing three consistency properties is presented. This model provides a conceptual framework for devising schemes and algorithms to solve the inconsistency problems of divergence, causality violation, and intention violation. This model was originally developed in the context of a collaborative text editing system called REDUCE, with the view that this model is generic and applicable to other editing environments. Collaborative graphics editing systems
also suffer from these three inconsistency problems (refer to Sun et. al. [108, 105] for inconsistency problems in REDUCE). Therefore, this consistency model is also applicable to collaborative graphics editing systems.

Following Lamport [57], a causal (partial) ordering relation on operation is defined. It is based on operation's generation and execution sequences as defined in Definition 1.1. Then, by using causal ordering, a dependency relationship between operations is defined in Definition 1.2.

**Definition 1.1.** *Causal ordering relation ‘→’*

Given two operations \( O_a \) and \( O_b \), generated at sites \( i \) and \( j \), then \( O_a \rightarrow O_b \), iff:

1. \( i = j \) and the generation of \( O_a \) happened before the generation of \( O_b \), or

2. \( i \neq j \) and the execution of \( O_a \) at site \( j \) happened before the generation of \( O_b \),

3. there exists an operation \( O_x \), such that \( O_a \rightarrow O_x \) and \( O_x \rightarrow O_b \). \( \square \)

**Definition 1.2.** *Dependent and independent operations*

Given any operations \( O_a \) and \( O_b \):

1. \( O_b \) is said to be *dependent* on \( O_a \) if and only if \( O_a \rightarrow O_b \);

2. \( O_a \) and \( O_b \) are said to be *independent* (or concurrent) if and only if neither \( O_a \rightarrow O_b \), nor \( O_b \rightarrow O_a \), which is expressed as \( O_a \parallel O_b \). \( \square \)
Definition 1.3. *Intention of an operation*

The intention of an operation $O$ is the execution effect which can be achieved by applying $O$ on the document state from which $O$ was generated.

Take note that, throughout this thesis, the *effect* of an operation refers only to the *syntactic* of operations. If an operation is to change the color of an object to black, then the syntactic effect of this operation is limited to change the color of that object to black. Operation’s effect does NOT cover the *semantic meaning* of the operation on a document. For example this object could be an apple, and it may be semantically incorrect to have a black apple. However, the syntactic effect of changing the color of the object to black is always correct no matter whether the result is sensible, or the operation was generated deliberately or accidentally. The syntactic effect of an operation is unambiguous. Consequently, the *intention* of an operation, as defined above, refers only to the syntactic effect of applying the operation on a particular document state, i.e. the one from which the operation was originally generated.

Definition 1.4. *A consistency model*

A collaborative editing system is said to be consistent if it always maintains the following properties:

1. *Convergence*: When the same set of operations have been executed at all sites, all copies of the shared document are identical.

2. *Causality preservation*: For any pair of operations $O_a$ and $O_b$, if $O_a \rightarrow O_b$, then $O_a$ is executed before $O_b$ at all sites.
3. *Intention preservation:* For any operation \( O \), the effects of executing \( O \) at all sites are the same as the intention of \( O \), and the effect of executing \( O \) does not change the effects of independent operations. \( \Box \)

In essence, the *convergence* property ensures the consistency of the final results at the end of a collaborative editing session; the *causality preservation* property ensures the consistency of the execution orders of dependent operations during a collaborative editing session; and the *intention preservation* property ensures:

1. that the effect of executing an operation at remote sites achieves the same effect as executing this operation at the local site at the time of its generation; and
2. that the execution effects of independent operations do not interfere with each other.

This consistency model imposes an execution order constraint on dependent operations only, but leaves it open for execution order of independent operations as long as the convergence and intention preservation properties are maintained. The consistency model effectively specifies, on the one hand, what assurance a collaborative editing system promises to its users and, on the other, what properties the underlying consistency maintenance mechanisms must support.

**Concurrency control techniques**

This section first examines existing concurrency control techniques to determine which technique can be used to preserve which consistency property. This is followed by detailed discussion of schemes for maintaining causality preservation and convergence.
Overview of existing approaches

The following concurrency control techniques have been used in database, distributed, and collaborative editing systems:

- **Turn-taking.** With turn-taking only one user at a time has the ‘token/floor’ to edit the shared document [32]. Access to the token may be controlled by internal technical protocols (implemented by software) or through external social protocols (followed by users collaborating to mediate their actions [42]). Turn-taking is application independent, therefore, it has been used in application sharing programs (when an editor is shared, then it is a collaborative editing system), such as Microsoft’s NetMeeting [70], Hewlett-Packard’s SharedX [59], and XShare [36].

With turn-taking, there is only one active user at any given time, so, in such systems none of the three inconsistency problems occurs. Therefore, convergence, causality preservation, and intention preservation are ensured at the price of not supporting concurrent editing. Consequently, this approach is limited to situations where a single active user fits into the needs of collaborative working, and is ill suited to application environments where the nature of collaboration is characterized by concurrent streams of activities from multiple users [28] as required by GRACE.

- **Locking.** An object is first locked before it is updated, so only one user at a time is able to update an object. Concurrent editing is allowed under the condition that multiple users are locking and editing different objects. Locking can prevent multiple users from generating conflict editing operations on the
same object, since only one user can edit one object at any given time. Examples of such systems include: Aspects [113], CoDiagram [10], Ensemble [76], GroupGraphics [84], ShrEdit [69], and SASSE [3].

Locking is application dependent. In graphics editing systems, non-conflicting operations are commutative. They can be executed in any order without causing divergence, and their effects can all be accommodated without loss. Locking prevents the generations of conflicting operations, hence it resolves intention violation and intra-object divergence problems but not inter-object divergence problems. However, in the text environment, the occurrence of divergence and intention violation is independent of whether or not editing operations refer to the same object. So locking does not resolve these two problems [107]. Furthermore, for any type of application, locking does not resolve causality violation because this problem is application independent. Locking also increases the response time due to the necessity of requesting and releasing locks in distributed environments.

- **Serialization.** With the serialization approach, operations may be generated concurrently, but the execution effects will be the same as if all operations were executed in the same serial order at all sites. This approach is also known as the *timestamp ordering* approach because the ordering of operation is determined by the timestamp on the operation. Serialization can be achieved either by pessimistically delaying the execution of an operation until all totally preceding operations have been executed [57, 106] or by optimistically executing operations upon their generation or arrival [42, 55]. Examples of systems using optimistic serialization are: CAB [61], GroupDesign [55], GroupDraw [43] and
LICRA [53].

Serialization can solve the problem of both intra-object and inter-object divergence. However, serialization-based protocols have the following problems. First, the intention violation problem cannot be solved by serialization. Second, the causality violation problem remains unsolved if operations are executed upon their arrival in the optimistic case or if the serial order is inconsistent with the causal ordering among operations in the pessimistic case. Lastly, responsiveness may be lost if operations are not executed immediately after their generation, but after some delay as in pessimistic serialization.

- **Causal ordering.** Operations may be generated and executed concurrently, but their execution order is constrained by their natural causal order. This can be achieved by using the well-known *vector logical clock* [34, 88]. In this approach, local operations can be executed immediately after their generation, so responsiveness is good; but some remote operations may be delayed until all causally preceding operations have been executed [31, 105]. This approach can achieve causality preservation only, but it does not address the problem of divergence and intention violation. Therefore, in many collaborative editing systems, this approach is used in combination with other concurrency control techniques.

- **Transformation.** Operations can be generated and executed concurrently, but they may be transformed before their execution so that the execution of the same set of properly transformed operations in different orders could produce identical document states [31]. Transformation is often combined with
the causal-ordering approach to achieve both convergence and causality preservation. In this approach, local operations can be executed immediately after their generation, so responsiveness is good. Transformation is application dependent. It has been used to achieve intention preservation in collaborative text editing systems, e.g. GROVE [31], Joint Emacs [90] Jupiter [77], and REDUCE [14, 101, 104, 105, 103]. However, transformation cannot solve the intention violation problem caused by conflicting operations in collaborative graphics editing systems.

From the above analysis of concurrency control methods, one can conclude that in collaborative graphics editing systems:

1. divergence (for both intra-objects and inter-objects) can be solved by optimistic serialization while satisfying the three requirements of real-time, distributed and, unconstrained editing; and

2. causality violation can be solved by imposing causal ordering which also satisfy these three requirements; however,

3. there exists no concurrency control method that can solve intention violation caused by conflicting operations while satisfying all three requirements.

Since the existing solutions for causality violation (by imposing causal ordering) and divergence (by serialization) satisfy the requirements for GRACE, these schemes will be presented and their applicability to GRACE examined. These schemes were originally developed for the collaborative text editing system, REDUCE [105, 108].
A causality preservation scheme

In order to capture the causal relationship among all operations in a system, a timestamping scheme based on a data structure called State Vector (SV), can be used [31, 105]. Let $N$ be the number of collaborating sites in the system. Assume that sites are identified by integers $0, \ldots, N-1$. Each site maintains a $SV$ with $N$ components. Initially, $SV[i] := 0$, for all $i \in \{0, \ldots, N-1\}$. After executing an operation generated at site $i$, $SV[i] := SV[i] + 1$. An operation is executed at the local site immediately after its generation and then multicast to remote sites with a timestamp of the current value of the local $SV$.

Definition 1.5. Conditions for executing remote operations

Let $O$ be an operation generated at site $s$ and timestamped by $SV_0$. $O$ is causally ready for execution at site $d$ (where $d \neq s$) with a state vector $SV_d$ only if the following conditions are satisfied:

1. $SV_0[s] = SV[s] + 1$, and

2. $SV_0[i] \leq SV_d[i]$, for all $i \in \{0, 1, \ldots, N-1\}$ and $i \neq s$. □

The first condition ensures that $O$ must be the next operation in sequence from site $s$, so no operation originated at site $s$ has been missed by site $d$. The second condition ensures that all operations originated at other sites and executed at site $s$ before the generation of $O$ must have been executed at site $d$. Altogether, these two conditions ensure that all operation which causally precede $O$ have been executed at site $d$. It can be shown that if a remote operation is executed only when it satisfies the above two conditions, then all operations will be executed in their causal orders,
thus achieving the causality preservation property of the above consistency model.

A convergence scheme

The causality-preservation scheme imposes causally ordered execution only for dependent operations and allows an operation to be executed at the local site immediately after its generation (for achieving good responsiveness). This implies that the execution order of independent operations may be different at different sites. A question arises: how to ensure the convergence property in the presence of different execution order of independent operations? To solve this problem, first a total ordering relation among operations is defined as follows.

**Definition 1.6.** Total ordering relation ‘⇒’

Given two operations $O_a$ and $O_b$, generated at sites $i$ and $j$ and timestamped by $SV_{O_a}$ and $SV_{O_b}$ respectively, then $O_a ⇒ O_b$, if and only if:

1. $\text{sum}(SV_{O_a}) < \text{sum}(SV_{O_b})$, or
2. $i < j$ when $\text{sum}(SV_{O_a}) = \text{sum}(SV_{O_b})$,

where $\text{sum}(SV) = \sum_{i=0}^{N-1} SV[i]$. $\square$

It can be shown that the total ordering relation ‘⇒’ is consistent with the causal order relation ‘→’ in the sense that if $O_a \rightarrow O_b$, then $O_a ⇒ O_b$ [109].

In addition, each site maintains a history buffer (HB) for saving executed operations at each site. Based on the total ordering relation and the history buffer, the following undo/redo scheme is defined.
Algorithm 1.1. The undo/do/redo scheme

When a new operation $O_{new}$ is causally ready, the following steps are executed:

1. **Undo** operations in $HB$ which totally follow $O_{new}$ to restore the document to the state before their execution.

2. **Do** $O_{new}$.

3. **Redo** all operations that were undone from $HB$.

One assumption made by the above scheme is that all operations in the collaborative editing system are reversible. It is required that buffered operations should contain enough information in order to be undone and redone. It should be noted that the undo operation involved in the undo/do/redo scheme is only an internal or system initiated operation, rather than an external operation initiated by the user. Therefore, the implementation of the undo/do/redo scheme should only display the final result on the user interface, instead of all intermediate results produced in the undo/do/redo process. It can be shown that under the undo/do/redo scheme, when all sites have executed the same set of operations, the editing effect will be the same as if all operations where executed in the total order `$\Rightarrow$' at all sites, thus ensuring the convergence property [109].

*Integrating concurrency control schemes*

The three inconsistency problems can be solved by either a single concurrency control scheme (which is capable of doing so) or a number of schemes that can be integrated to work together. The two schemes presented above have been shown working together
with an operational transformation (intention preservation) scheme to solve all three problems in REDUCE [105]. However, it needs to be determined if these two schemes can work with an intention preservation scheme in GRACE.

Both convergence and intention preservation are concerned with the result/effect of operation execution. Whereas causality preservation is concerned only with the order of operation execution. The causality preservation scheme described above is independent of the operation’s effect. Hence, it can work with any convergence and intention preservation schemes. Therefore, it is directly adopted for GRACE. Any GRACE operation can be executed only if it is causally ready.

The effect produced by a convergence scheme should be consistent with the effect produced by the intention preservation scheme. Since the intention preservation scheme has not yet been determined, therefore, whether the above convergence scheme can be adopted for GRACE needs to be examined. If it cannot, then new convergence and intention preservation schemes will need to be devised.

1.4 Summary of contributions

This dissertation focuses on maintaining consistency in real-time object-based collaborative graphics editing systems and has made the following contributions to the CSCW field.

Maintaining consistent execution effect by conflict resolution

Execution of conflicting operations in collaborative graphics editing systems may produce inconsistent results. A multiple object version scheme [17, 102] to maintain consistency under such conditions is proposed. With this scheme, versions of the
same object are created, one for each conflict operation. The effect of each conflicting operation is preserved on a different version. From the result produced, the users can make an informed decision on which operations effect they wish to keep. An object identification scheme is proposed to work in conjunction with the multiple object version scheme. This identification scheme is able to uniquely and consistently identify objects as well as being able to determine versions of the same object.

Maintaining consistent undo effect of any operation

Undo is the standard feature in most single-user editors to undo executed operations. Proposed is an Any Undo scheme [22] which allows users to undo any executed operation at any time in collaborative graphics editing systems. The unique contributions to undo in collaborative graphics editing systems include:

1. defining the undo effect for undoing any executed graphics operation;

2. designing algorithms to produce the required undo effect; and

3. designing algorithms to maintain consistency by removing and recreating object versions.

Maintaining consistency by conflict prevention

Locking can be used to prevent the generation of conflicting operations. However, relying on locking to maintain consistency introduces the overhead of the time to obtaining a lock before an operation can be generated. Hence, the response time is slowed. Proposed is an instant optional locking scheme [19, 21] in collaborative graphics editing systems for preventing the generation of conflicting operations. This
locking scheme allow users to lock place locks only when required, and the locks are obtained without network synchronization delay.

1.5 Thesis outline

This thesis focuses on the design issue of consistency maintenance for collaborative graphics editing systems. The rest of this thesis is organized as follows. Chapter 2 discusses how to resolve inconsistency problems caused by the execution of conflicting operations, by using the technique of multiple object version scheme. Chapter 3 examines how to undo any executed operations while maintaining consistency. Chapter 4 discusses how to prevent the generation of conflicting operations by using optional locking. Chapter 5 presents some important GRACE implementation issues. Chapter 6 presents the conclusion and planned future work.
Chapter 2

Maintaining consistent execution effect by conflict resolution

How to maintain consistent execution effect on copies of the shared document? As discussed in the previous chapter, the inconsistency problem which needs to be solved is intention violation caused by the execution of conflicting operations. In this chapter a multiple object version scheme is proposed to resolve conflict and preserve the intention of operations.

2.1 Operation conflicts and multiple versions

2.1.1 Conflict and compatible relations

In the graphics editing environment, if all operations are generated causally after each other, then simply executing the operations in their causal orders will preserve the intentions of all operations. In a group editing environment, however, operations may be generated concurrently by multiple users. If all concurrent operations are targeting different objects, then they can be executed in any order without violating their intentions. However, concurrent operations may target the same object and may conflict with each other. For example, suppose user 1 generates $O_1 = \text{Move}(G, X)$ to
move object $G$ to position $X$, and user 2 concurrently generates $O_2 = Move(G,Y)$ to move $G$ to position $Y$, where $X \neq Y$. Both operations will be executed at their local sites immediately to give a quick response, and then propagated to the other sites for execution. Since $O_1$ and $O_2$ are moving the same object $G$ to two different positions, it is impossible to accommodate their conflicting effects in the same target object. In general, two concurrent operations are in conflict if they are targeting the same object but changing the same attribute to different values.

To give a precise definition of operation conflict, the following notations are first introduced:

1. $Target(O)$ denotes the target object of operation $O$;

2. $Att.Type(O)$ denotes the attribute type of $O$; and

3. $Att.Value(O)$ denotes the attribute value of $O$.

**Definition 2.1.** Conflict relation `$\otimes$`

Given two operations $O_1$ and $O_2$, they conflict with each other, i.e., $O_1 \otimes O_2$, iff

1. $O_1 \parallel O_2$;

2. $Target(O_1) = Target(O_2)$;

3. $Att.Type(O_1) = Att.Type(O_2)$; and

4. $Att.Value(O_1) \neq Att.Value(O_2)$. 

In contrast, if a pair of operations are not conflicting, then they are compatible, as defined below.
Definition 2.2. Compatible relation ‘○’

Given two operations $O_1$ and $O_2$, if they do not conflict with each other, they are compatible, expressed as $O_1 \circ O_2$. □

Create operations will always be compatible with each other because each Create operation creates a different object. Create operations do not conflict with other types of operations because operations targeting the same object as a Create operation must be causally after this Create. Concurrent Destroy operations targeting the same object are compatible with each other because they all have the same intended effect i.e. to remove the object. Destroy operations do not conflict with update operations because Destroy operations do not change attribute values\(^1\). Hence, only update operations of the same type can conflict with each other.

2.1.2 Accommodating all operation effects

For compatible operations, if they are targeting the same object, they can be applied to the same object. For conflicting operations, such as $O_1 = \text{Move}(G, X)$ and $O_2 = \text{Move}(G, Y)$, what combined effects could they have without violating their intentions?

One possible combined effect is the null-effect, which means neither of the conflicting operations has a final effect on the target object. This can be achieved by rejecting/undoing an operation when it is found to be conflicting with another operation, as shown in Fig 2.1. The final results at both sites are identical (empty). However, this null effect does not preserve the intentions of the two operations since

\(^1\)To be more precise, Destroy operations do not change the same attribute as update operations because a special attribute is introduced (see Chapter 3) that is changed by Destroy.
neither operation has an effect at the remote site and the effect of one operation has been undone by another independent operation. The consequence of this intention violation is that, whenever there is a conflict, the work concurrently done by involved users will be destroyed, which is highly undesirable in the collaborative working environment. Moreover, when a conflict occurs, the involved users are provided with no explicit information about what the other users actions were or intentions might be, and hence may not be able to take proper actions to resolve the conflict.

The second possible combined effect is the single operation effect, which is to retain the effect of only one operation, either $O_1$ or $O_2$. This can be achieve by enforcing a serialized effect among all operations. As shown in Fig. 2.2, when $O_2$ arrives at User 1, it moves $G$ to position $Y$ (effectively undoing the effect of $O_1$); when $O_1$ arrives at User 2, it is rejected. The final results at both user’s sites are identical. However, this single operation effect violates the intentions of both operations since
Figure 2.2: *Single operation effect for conflicting operations*

one operation ($O_1$) has no effect at User 2, and the other operation ($O_2$) has changed the effect of an independent operation ($O_1$) at User 1. One consequence of this intention violation is that whenever there is a conflict, only one user's work can be preserved. Another consequence is that users are not ensured to see the effects of the same set of operations: e.g. User 1 sees the effects of both $O_1$ and $O_2$, but User 2 never sees the effect of $O_1$. Generally, when there are multiple conflicting operations, each user may see the effects of an arbitrary number of operations, depending on the order in which operations arrive at each site. Therefore, when a conflict occurs, users may not see a consistent and explicit picture about what other users intended to do, and hence they may not be able to take proper actions to resolve their conflict. Although the system could notify the involved users that there is a conflict, not all users are provided with the same amount of explicit information about what users intended to do.
Figure 2.3: All operations effect for conflicting operations

To preserve all work concurrently produced by multiple users in the face of conflicts, proposed is an all operations effect, based on a multiple versions strategy \cite{17, 102, 100}\footnote{Earlier work which lead to the design of the current solution can be found in \cite{15, 23, 18}}: two versions of $G$, $G_1$ and $G_2$, will be created, with $O_1$ and $O_2$ being applied to $G_1$ and $G_2$, respectively. In this way, the effects of both operations are accommodated in two separate versions, as shown in Fig. 2.3.

This all operations effect preserves the intentions of both operations since the effects of $O_1$ and $O_2$ at their local sites are the same as their effects at the remote sites and they do not change the effects of each other. With the all operations effect, the system is able to ensure that the work produced by all users will always be retained regardless whether there is a conflict or not. The only side effect of this approach is that the single version object may be converted to multiple versions if a conflict occurs. The system could notify the users that there is a conflict, e.g. by highlighting
the multiple versions of the same object. Since all users are provided with a consistent and explicit picture about what other users intended to do, they could make better assessment of the situation and may decide to keep one of the versions or even all of them if that is desired.

It is worth pointing out that a similar all-operations-effect strategy has also been used in the collaborative text editing domain [105, 103]: when there are two concurrent Insert operations inserting two strings, $S_1$ and $S_2$, at the same position, even if $S_1$ is a substring of $S_2$, both strings are maintained in the document (one after the other) rather than being merged into one. In general, a groupware design principle is advocated: In the face of a conflict, it is usually better to preserve and display all users’ work to facilitate a user-decided solution to the conflict, rather than to destroy or hide users’ work to impose a system-decided solution to the conflict. Because it is generally unfeasible for the system to have the knowledge to properly resolve conflicts among concurrent users, conflicts are best resolved by collaborative users, with the system providing explicit information about other users’ actions.

2.1.3 Combined effect rules

Given a group of $N$ operations targeting the same object, if they are all mutually compatible with each other, then they can be applied to the original target object without creating new versions; and if they are all mutually conflicting with each other, then $N$ versions will be created to accommodate each operation’s effect in a separate version according to the multiple object version strategy proposed in the previous section. However, if there is a mixture of compatible and conflicting operations in the group, it becomes non-trivial to determine how many versions to create, and how to apply which operations to which versions. In the following discussion, the notation
$G\{O_x\}$ will be used to represent an object $G$ with the effect of $O_x$ and $G\{\}$ represents the initial state.

To start with, consider a simple scenario with three operations: $O_1$, $O_2$, and $O_3$. Suppose they are targeting the same object $G$, and their mutual conflict relations are: $O_1 \otimes O_2$, $O_1 \otimes O_3$, and $O_2 \otimes O_3$. What combined effects should these three operations have?

Since $O_1 \otimes O_2$, they must be separately applied to two versions $G\{O_1\}$ and $G\{O_2\}$ according to the multiple versions strategy. In general, we have the following combined effect rule:

**Combined Effect Rule 1 (CER1):** Given two operations, $O_1$ and $O_2$, targeting object $G$. If $O_1 \otimes O_2$, they must be applied to different versions, $G\{O_1\}$ and $G\{O_2\}$, made from $G$. This means $O_1$ and $O_2$ can never be applied to the same object.

The question is: how to combine $O_3$’s effect? One possibility is to make a separate version, $G\{O_3\}$. The problem with this approach is that it unnecessarily creates two versions, $G\{O_2\}$ and $G\{O_3\}$, for two compatible operations. To avoid unnecessary versions, it is proposed that two compatible operations, $O_2$ and $O_3$, are combined in a common version, $G\{O_2, O_3\}$. In general, to minimize the number of versions for an object, the following combined effect rule is used to justify the creation of different versions.

**Combined Effect Rule 2 (CER2):** Given any two versions, $G_1$ and $G_2$, made from the same object, $G$, there must be at least one operation, $O_1$, applied to $G_1$,
and at least one operation, $O_2$, applied to $G_2$, such that $O_1 \odot O_2$.

Furthermore, consider another scenario with three operations: $O_1$, $O_2$, and $O_3$, targeting the same object, $G$. Suppose their mutual conflict relations are: $O_1 \odot O_2$, $O_1 \odot O_3$, and $O_2 \odot O_3$. Since $O_1 \odot O_2$, two versions, $G\{O_1\}$ and $G\{O_2\}$, need to be created according to CER1. The question is: which one of the two versions should $O_3$ be applied to? In other words, which one of two compatible operations should $O_3$ be combined with?

One possibility is to combine $O_3$ with either $O_1$ (i.e. $G\{O_1, O_3\}$) or $O_2$ (i.e. $G\{O_2, O_3\}$), chosen by the system (randomly or by using their total ordering). This approach does not produce any unnecessary version (according to CER2), but may have an abnormal phenomenon at the user interface, as shown in Fig. 2.4.

![Diagram showing the operations and their conflicts](image)

**Figure 2.4: A scenario for motivating CER3**

Suppose the system has chosen to combine $O_3$ with $O_1$. At the site of User 3, the
following abnormal phenomenon occurs: \( O_3 \) is first applied to its target object, \( G \), to produce \( G\{O_3\} \); then \( O_2 \) arrives and is combined with \( O_3 \) to produce \( G\{O_3, O_2\} \) since they are compatible (User 3 has no knowledge about \( O_1 \) at this stage); finally \( O_1 \) arrives and is found to be conflicting with \( O_2 \), so \( O_3 \) has to be undone to produce \( G\{O_2\} \), and then redone in a new version to produce \( G\{O_3, O_1\} \) (to achieve the system’s chosen combined effect). In this scenario, User 3 will observe that \( O_3 \)’s effect is changing from one version to another, due to the inconsistency between its initial effect and its final effect. This abnormal effect is undesirable, and also violates the intentions of operations since one operation (e.g. \( O_1 \)) changes (by undoing) the effect of another independent operation (e.g. \( O_3 \)). It should be pointed out that no matter which combined effect (\( O_3 \) combined with \( O_1 \) or \( O_2 \)) the system chooses, at least one user (User 1 or User 3) in this scenario will observe that the original execution effect of \( O_3 \) is undone and then redone in another object.

To avoid this abnormal interface effect, we propose to combine \( O_3 \) with both \( O_1 \) and \( O_2 \) to produce \( G\{O_1, O_3\} \) and \( G\{O_2, O_3\} \). In this way, no matter in which order these three operations are executed, the final combined effect will be the same, \( \{O_1, O_3\} \) and \( \{O_2, O_3\} \), at all sites, without any abnormal interface effect. In general, we have the following additional rule to determine the combined effects of compatible operations in the face of mixed compatible and conflict operations.

**Combined Effect Rule 3 (CER3):** Given any group of operations, if they are mutually compatible and target the same object, then their effects must be combined in at least one common object version of the target object.
In summary, **CER1**, **CER2** and **CER3** are the three criteria for judging whether a combined effect for a group of operations targeting the same object is correct or not. By applying these criteria, the following combined effects can be achieved:

1. conflicting operations are accommodated in different versions;

2. compatible operations are combined in common versions;

3. there is at least one pair of conflicting operations between any pair of versions; and

4. there is at least one version combining the effects of any group of compatible operations.

### 2.2 Combined effects for any group of operations

In the previous section, simple scenarios have been used to derive and illustrate the criteria (i.e. **CER1**, **CER2** and **CER3**) to determine the combined effects of conflicting and compatible operations. However, in a concurrent real-time collaborative editing environment, a group of operations may have rather arbitrary and complex conflict relationships among them. A major technical problem here is: given an arbitrary group of operations targeting the same object, how to determine their combined effect, which is complying with **CER1**, **CER2** and **CER3**?

#### 2.2.1 Conflict relation matrix and triangle

To solve this problem, first introduce the *conflict relation matrix* to capture the complete picture of conflict relationships among any group of operations targeting the same object.
Given a group of $n$ operations, $O_1, O_2, ..., O_n$, targeting the same object, their conflict relationships can be fully and uniquely expressed by a $n \times n$ Conflict Relation Matrix (CRM), in which element $CRM[i, j]$, $1 \leq i, j \leq n$ is filled with ‘⊗’ if $O_i \otimes O_j$, otherwise it is filled with ‘○’. For example, a $3 \times 3 CRM$ for three operations is shown in Fig. 2.5(a).

\[
\begin{array}{ccc}
\text{OP} & O_1 & O_2 & O_3 \\
O_1 & & ⊗ & ⊗ & ⊗ \\
O_2 & ⊗ & & ⊗ & ⊗ \\
O_3 & ⊗ & ⊗ & & ⊗ \\
\end{array}
\]

(a) CRM

\[
\begin{array}{ccc}
\text{OP} & O_1 & O_2 & O_3 \\
O_1 & & ⊗ & \\
O_2 & ⊗ & & \\
O_3 & ⊗ & & \\
\end{array}
\]

(b) CRT

Figure 2.5: CRM versus CRT

Since ⊗ and ○ relations are symmetric (i.e., $CRM[i, j] = CRM[j, i]$), and an operation is always compatible with itself (i.e., $CRM[i, i] = ○$), by omitting these redundant and constant relation elements, the conflict matrix can be compressed to a $(n - 1) \times (n - 1)$ Conflict Relation Triangle (CRT). For example, the $3 \times 3 CRM$ in Fig 2.5(a) can be compressed into an equivalent $2 \times 2 CRT$ in Fig. 2.5(b).

2.2.2 Compatible groups set

An alternative way of expressing the conflict/compatible relationships for a group of operations is called Compatible Groups Set (CGS), which is defined as follows.

**Definition 2.3.** Compatible groups set

Given a group of operations, $GO$, its corresponding Compatible Groups Set (CGS) is expressed as follows:
$CGS = \{CG_1, CG_2, ..., CG_n\}$

where $CG_i = \{O_1, O_2, ..., O_k\}$, and

1. all operations in any $CG_i$ must be mutually compatible;

2. for any operation $O \in GO$, there must be at least one $CG_i \in CGS$, such that $O \in CG_i$; and

3. for any pair of operations $O_x, O_y \in GO$, if $O_x \circ O_y$, there must be at least one $CG_i \in CGS$, such that $O_x, O_y \in CG_i$. $\square$

For example, the conflict relation expressed by the CRT in Fig. 2.5(b) can also be captured by: $CGS = \{\{O_1, O_3\}, \{O_2, O_3\}\}$. In general, given a CRT, a CGS can be derived from it by using the following algorithm.

**Algorithm 2.1.**

Given a CRT for a group of $N$ operations $GO$, a CGS corresponding to this CRT can be obtained as follows:

1. $CGS = \{\};$

2. For $1 \leq i \leq N - 1$, and $i < j \leq N$
   
   If $CRT[i, j - 1] = \circ$
   
   Then $CGS = CGS + \{\{O_i, O_j\}\};$

3. For $1 \leq i \leq N$,
   
   If $O_i \notin CG_k$ for all $k \in \{1, 2, ..., |CGS|\}$
Then $CGS = CGS + \{\{O_i\}\}$.

For example, as shown in Fig. 2.6, $\{O_2, O_3\} \in CGS$ since $CRT[2, 3 - 1] = \odot$, and $\{O_i\} \in CGS$ since $O_i$ is not in any other $CG$ in $CGS$.

\[
\begin{array}{|c|c|c|}
\hline
OP & O_2 & O_3 \\
\hline
O_1 & \otimes & \otimes \\
O_2 & - & \otimes \\
\hline
\end{array}
\]

$CGS = \{\{O_i\}, \{O_2, O_3\}\}$

Figure 2.6: A CRT and its corresponding CGS

It should be noted that in the $CGS$, the compatible relationships among operations are \textit{explicitly} expressed by their co-existence in at least one $CG$. However, the conflict relationships among operations are \textit{implicitly} expressed by their non-coexistence in any $CG$.

\section{2.2.3 Equivalent CGS}

If two compatible groups sets, $CGS_i$ and $CGS_j$, capture the same compatible relationships for the same group of operations, then they are \textit{equivalent}, denoted as $CGS_i \equiv CGS_j$. There exist some transformation rules which can be used to transform a $CGS$ into another equivalent $CGS$.

In the following, the notation $CG_i \odot CG_j$ is used to mean that all operations in both $CG_i$ and $CG_j$ are mutually compatible.
Rule 1.

Given a $CGS$, for any pair $CG_i, CG_j \in CGS$, if $CG_i \subsetneq CG_j$, $CG_j \subsetneq CG_i$, and $CG_i \cap CG_j$, then $CGS \equiv CGS - \{CG_i, CG_j\} + \{CG_i \cup CG_j\}$. □

Rule 1 states that if neither of the groups embraces the other (non-embracing groups) and all operations in the two groups are mutually compatible (mutually-compatible groups), then these two groups can be replaced by their union. This rule can be extended to any $m (> 2)$ non-embracing but mutually-compatible groups. With this rule, multiple small groups can be merged into a single big group which includes all mutually compatible operations.

Rule 2.

Given a $CGS$, if there exist $CG_i, CG_j \in CGS, i \neq j$, such that $CG_i \subseteq CG_j$, then $CGS \equiv CGS - \{CG_i\}$. □

Rule 2 states that if one group is a subgroup of another group in a $CGS$, then the subgroup can be removed.

2.2.4 Normalized CGS

We are particularly interested in a special form of $CGS$, called $Normalized\ Compatible\ Groups\ Set\ (NCGS)$, which is defined below.
**Definition 2.4.** Normalized compatible groups Set

Given a CGS for any group of operations, GO, targeting the same object, the CGS is a Normalized CGS (NCGS), iff:

1. for any group of mutually compatible operations in GO, there must be at least one CG ∈ CGS, such that all these compatible operations coexist in CG; and

2. for any pair CGi, CGj ∈ CGS, there must be at least one Ox ∈ CGi, and one Oy ∈ CGj, such that Ox ⊗ Oy. □

By using Rules 1 and 2, a CGS can always be transformed into a NCGS. The following algorithm can be used to obtain a NCGS from a given CRT for any group of operations targeting the same object.

**Algorithm 2.2.**

Given a CRT for a group of operations, a NCGS corresponding to this CRT can be obtained as follows:

1. Obtain a CGS for this CRT by using Algorithm 2.1.

2. Apply Rule 1 to transform CGS into CGS′, so that all non-embracing but mutually-compatible groups are replaced by their unions.

3. Apply Rule 2 to transform CGS′ into CGS″, so that all subgroups are removed.

4. Return NCGS = CGS″.
\begin{center}
\begin{tabular}{|c|c|c|c|}
  \hline
  OP & $O_2$ & $O_3$ & $O_4$ \\
  \hline
  $O_1$ & $\otimes$ & $\otimes$ & $\otimes$ \\
  \hline
  $O_2$ & $\otimes$ & $\otimes$ & $\otimes$ \\
  \hline
  $O_3$ & $\otimes$ & $\otimes$ & $\otimes$ \\
  \hline
\end{tabular}
\end{center}

$$CGS = \{\{O_1, O_3\}, \{O_1, O_4\}, \{O_2, O_3\}, \{O_2, O_1\}, \{O_3, O_4\}\}$$
\[= \{\{O_1, O_3, O_4\}, \{O_2, O_3, O_4\}, \{O_3, O_4\}\} \text{ (by Rule 1)}
\[= \{\{O_1, O_3, O_4\}, \{O_2, O_3, O_4\}\} \text{ (by Rule 2)}
\[= \text{NCGS}$$

Figure 2.7: A CRT, and its corresponding CGS and NCGS

An example of applying Algorithm 2.2 to transform a CGS into a NCGS is given in Fig 2.7.

The following theorem establishes the uniqueness property of the NCGS.

\textbf{Theorem 1.}

Given a group of operations, GO, targeting the same object, the NCGS for this GO is unique.

\textbf{Proof:} Suppose there are two NCGSs, NCGS$_1$ and NCGS$_2$, for the same GO. First, prove that for $CG_x \in NCGS_1$, there must exist a $CG_y \in NCGS_2$, such that $CG_x = CG_y$. Since both NCGS$_1$ and NCGS$_2$ are for the same GO, all operations in $CG_x$ of NCGS$_1$ must also be in NCGS$_2$. Moreover, since all operations in $CG_x$ are mutually compatible, they must all be in at least one compatible group, $CG_y$, in NCGS$_2$ according to Condition (1) of Definition 2.4. Furthermore, it is impossible for $CG_y$ to contain one extra compatible operation, $O_y$. Otherwise, there must be at least one compatible group, $CG_x'$, in NCGS$_1$, which contains both $O_y$ and all operations
in $CG_x$ according to Condition (1) of Definition 2.4. Then, $CG_x$ must be subgroup of $CG'_x$, which is in contradiction to Condition (2) of Definition 2.4. Thus, $CG_x$ and $CG_y$ must contain the same group of compatible operations and hence $CG_x = CG_y$. By the same reasoning, it can be proven that for any $CG_y \in NCGS_2$, there must exist a $CG_x \in NCGS_1$, such that $CG_x = CG_y$. Thus the theorem follows.  

\[ \square \]

2.2.5 Combined effect specified by NCGS

The significance of the $NCGS$ is that it gives a formal specification of the combined effect for any group of operations targeting the same object.

Definition 2.5. NCGS specified combined effect

Given the $NCGS$ for a group of operations, $GO$, targeting object $G$, the combined effect for $GO$ is as follows:

1. For each $CG \in NCGS$, there is one object version made from $G$.

2. For all operations in the same $CG$, they will be applied to the same version corresponding to the $CG$.  

The combined effect specified by the $NCGS$ is unique because the $NCGS$ for a $GO$ is unique. Furthermore, the following theorem establishes that this combined effect complies with $CER_1$, $CER_2$, and $CER_3$.

Theorem 2.

The combined effects specified by $NCGS$ satisfy $CER_1$, $CER_2$ and $CER_3$.  

Proof:

1. For any pair of operations, $O_1$ and $O_2$, in the $NCGS$, if $O_1 \otimes O_2$, they could never coexist in the same $CG$ in the $NCGS$ according to Condition (1) of Definition 2.3. and hence they could never be applied to the same object version, which complies with CER1.

2. For any pair of compatible groups, $CG_i$ and $CG_j$, in the $NCGS$, there must be at least one $O_x \in CG_i$, and one $O_y \in CG_y$, such that $O_x \otimes O_y$, according to Condition (2) of Definition 2.4. Since there is one-to-one correspondence between the compatible groups in the $NCGS$ and the object versions made according to the $NCGS$ specified combined effect, CER2 is satisfied.

3. For a group of operations, if they are mutually compatible, they must coexist in at least one common $CG$ according to Condition 1 of Definition 2.4, so they will be combined in at least one common object version, which complies with CER3. \hfill \Box

In summary, the major result in this section is that, given a group of operations targeting the same object, their combined effect can be uniquely determined by the $NCGS$, and this combined effect complies with CER1, CER2, and CER3. The following sections will discuss how to achieve this unique and correct combined effect in a distributed, incremental, and consistent way.
2.3 Incremental creation of multiple versions

If the group of operations, GO, targeting the same object are all known in advance, the NC\textit{GS} for this GO can be constructed by using Algorithm 2.2; then multiple versions can be created and operations can be applied to proper versions according to the combined effects specified by the NC\textit{GS}. However, in real-time collaborative editing sessions, operations can be generated concurrently and may arrive at different sites in different orders. Because of high responsiveness considerations, it is not proper (or feasible) to postpone executing an operation until all other potentially concurrent operations have arrived. An operation should be allowed to execute as long as it is in the right causal order. This means that the system has to execute the group of operations one after another to incrementally create versions (if necessary) and combine the effects of all operations. In other words, a distributed algorithm is needed to incrementally construct the NC\textit{GS} at all sites.

Suppose a group of \textit{n} operations targeting the object arrive (and become causally ready for execution) at a site in the following order: \textit{O}_1, \textit{O}_2, ..., \textit{O}_n. The algorithm will construct a sequence of NC\textit{GS}s: \textit{NCGS}_1, \textit{NCGS}_2, ..., \textit{NCGS}_n in such a way that \textit{NCGS}_i is the NC\textit{GS} for the group of operations from \textit{O}_1 to \textit{O}_i, and the final \textit{NCGS}_n is the NC\textit{GS} for the whole group of operations. To achieve this, two technical problems need to be solved: one is how to apply operation \textit{O}_i on the \textit{NCGS}_{i-1} to produce \textit{NCGS}_i; and the other is how to identify all object versions corresponding to \textit{NCGS}_{i-1} at each step. The second problem will be addressed in the next section. In this section, a \textit{Multiple Object Versions Incremental Creation (MOVIC)} algorithm will be proposed to address the first problem.
2.3.1 The MOVIC algorithm

The following notations will be used in the description of the MOVIC algorithm:

1. \( O_i \) represents the \( i \)th operation to execute at any site.

2. \( NCGS_{i-1} \) represents the \((i - 1)\)th NCGS for operations from \( O_1 \) to \( O_{i-1} \).

3. \( NCGS_i \) represents the \( i \)th NCGS for operations from \( O_1 \) to \( O_i \).

4. \( O_i \odot CG \) means that \( O_i \) is compatible with all operations in \( CG \).

5. \( O_i \odot CG \) means that \( O_i \) is conflicting with all operations in \( CG \).

The objective of the MOVIC algorithm is to apply \( O_i \) to the \( NCGS_{i-1} \) (i.e. to add \( O_i \) to proper existing compatible groups in the \( NCGS_{i-1} \) or to create new compatible groups, if necessary) to produce the \( NCGS_i \).

Algorithm 2.3. \( MOVIC(O_i, NCGS_{i-1}) : NCGS_i \)

1. \( NCGS_i := \{ \} ; \; C := |NCGS_{i-1}| ; \)

2. Repeat until \( NCGS_{i-1} = \{ \} ; \)
   (a) Remove one \( CG \) from \( NCGS_{i-1} ; \)
   (b) If \( O_i \odot CG \), then \( CG := CG + \{ O_i \} ; \)
   (c) Else if \( O_i \odot CG \), then \( C := C - 1 ; \)
   (d) Else
      \begin{itemize}
      \item \( CG_{\text{new}} := \{ O | (O \in CG) \land (O \odot O_i) \} ; \)
      \item \( CG_{\text{new}} := CG_{\text{new}} + \{ O_i \} ; \)
      \end{itemize}
\[ NCGS_i^{\prime} := NCGS_i + \{ CG_{\text{new}} \}. \]

(e) \[ NCGS_i := NCGS_i + \{ CG \}; \]

3. If \( C = 0 \), then

(a) \( CG_{\text{new}} := \{ O_i \}; \)

(b) \( NCGS_i := NCGS_i + \{ CG_{\text{new}} \}; \)

4. For any \( CG_{\text{new}} \in NCGS_i \), if there is another \( CG \in NCGS_i \), such that \( CG_{\text{new}} \subseteq CG \), then \( NGCS_i := NCGS_i - \{ CG_{\text{new}} \}. \)

In the MOVIC algorithm, the \( NCGS_i \) is first initialized to an empty set, and \( C \) (a counter for the number of \( CGs \) which are not fully conflicting with \( O_i \)) is initialized to the size of the current \( NCGS_{i-1} \).

Then, \( O_i \) is checked in a loop against every \( CG \) in the \( NCGS_{i-1} \) one by one (Note: the order is not significant.) If \( O_i \) is compatible with all operations in \( CG \), then \( O_i \) is added to \( CG \), which means that \( O_i \) can be directly applied to that object version. Else if \( O_i \) is conflicting with all operations in \( CG \), then \( O_i \) is not added to \( CG \), which means \( O_i \) cannot be applied to that object version. In this case, the counter \( C \) is decremented. Otherwise, \( O_i \) must be partially compatible with some operations in \( CG \). In this case, a new group, \( CG_{\text{new}} \), is created which contains all operations in \( CG \) which are compatible with \( O_i \), and then \( O_i \) is added to \( CG_{\text{new}} \), which means \( O_i \) is applied to a new object version corresponding to \( CG_{\text{new}} \). The newly created \( CG_{\text{new}} \) and the existing \( CG \) (with possibly an additional \( O_i \)) are all added to the \( NCGS_i \). Since \( O_i \) is added only to groups with operations which are all compatible with \( O_i \), the resulting groups are ensured to be compatible groups (for Conditions 1 and 2
of Definition 2.3). Moreover, when $O_i$ is compatible with multiple operations in an existing group, it is always added to that group or a new group containing all these compatible operations. In this way, Condition 1 of Definition 2.4 is satisfied.

After checking all $CG$s in the $NCGS_{i-1}$, if $C = 0$, then $O_i$ must be either the first operation (i.e. $O_i = O_1$) or conflicting with all $CG$s in the $NCGS_{i-1}$. In this case, a new group, $CG_{new} = \{O_i\}$, is created (for Condition 2 of Definition 2.3).

A last but very important step in the MOVIC algorithm is to check each newly created group, $CG_{new}$, to see whether it is a subgroup of another group in the $NCGS_i$. If this is the case, $CG_{new}$ should be removed according to Rule 2 to ensure that there shall be at least one pair of conflicting operations in each pair of $CG$s in the new $NCGS_i$ (for Condition 2 of Definition 2.4).

Since creating a new compatible group corresponds to creating a new object version, and adding $O_i$ to an existing group or a new group corresponds to applying $O_i$ to the object version for that group, it is straightforward to derive the method of executing $O_i$ on the object versions corresponding to the $NCGS_{i-1}$ as follows:

1. If $O_i$ is added to an existing $CG$ in Step 2(b) of Algorithm 2.3, then $O_i$ is applied to the existing object version corresponding to $CG$.

2. If a $CG_{new}$ is created out of an existing $CG$ in Step 2(d), and this $CG_{new}$ is not removed in Step 4, then a new object version corresponding to $CG_{new}$ is created and $O_i$ is applied to it.

3. If a $CG_{new}$ with only $O_i$ is created in Step 3(a), then a new object version corresponding to the $CG_{new}$ is created and $O_i$ is applied to it.
2.3.2 Order independence property

The MOVIC algorithm has a very important property: no matter in which order a
group of \( n \) operations are processed, the final \( \text{NCGS}_n \) constructed by the MOVIC
algorithm is the same because there is only one unique \( \text{NCGS} \) for any group of op-
erations (see Theorem 1). This property is called *order independency*, which ensures
that a consistent final result can be achieved by executing the MOVIC algorithm
at all collaborating sites regardless of different operation execution orders. A formal
verification of this property is beyond the scope of this thesis. Some examples are
given below to illustrate the order independency property.

**Example 2.1.**

Given four operations, \( O_1, O_2, O_3, \) and \( O_4 \), with their conflict relationships ex-
pressed in Fig. 2.8\(^3\), consider the following two different execution orders:

<table>
<thead>
<tr>
<th>OP</th>
<th>( O_2 )</th>
<th>( O_3 )</th>
<th>( O_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O_1 )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
</tr>
<tr>
<td>( O_2 )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
</tr>
<tr>
<td>( O_3 )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
<td>( \otimes )</td>
</tr>
</tbody>
</table>

Figure 2.8: The CRT for Example 2.1

**Execution Order 1:** \( O_1, O_2, O_3, \) and \( O_4 \)

1. \( \text{NCGS}_1 = \{ \{ O_1 \} \} \)

\(^3\)It should be noted that the conflict relationships between \( O_i \) and operations in \( \text{NCGS}_{i-1} \) can be
detected on-the-fly by examining the state-vector timestamps and other parameters of operations.
The CRT is used here just to give a complete picture of the conflict relationships among all opera-
tions, which does not imply that a complete CRT for a group of operations has to be constructed
before applying the MOVIC algorithm.
2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$

3. $NCGS_3 = \{\{O_1\}, \{O_2\}, \{O_3\}\}$

4. $NCGS_4 = \{\{O_1, O_4\}, \{O_2, O_4\}, \{O_3, O_4\}\}$

**Execution Order 2:** $O_1$, $O_2$, $O_4$, and $O_3$

1. $NCGS_1 = \{\{O_1\}\}$

2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$

3. $NCGS_3 = \{\{O_1, O_4\}, \{O_2, O_4\}\}$

4. $NCGS_4 = \{\{O_1, O_4\}, \{O_2, O_4\}, \{O_3, O_4\}\}$
   \[
   \equiv \{\{O_1, O_4\}, \{O_2, O_4\}, \{O_4, O_3\}\} \quad \text{(by Rule 2)}
   \]

It can be seen that at Step 4 of Execution Order 2, $O_3$ is first checked against $\{O_1, O_4\}$ and a new group $\{O_4, O_3\}$ is created since $O_4 \oplus O_3$ but $O_1 \oplus O_3$; then $O_3$ is checked against $\{O_2, O_4\}$ and another (exactly the same) new group, $\{O_4, O_3\}$, is created for the same reason. However, one of the two new groups is removed according to Rule 2. In this way, the final $NCGS_4$ is the same for two different operation execution orders.

**Example 2.2.**

Given four operations, $O_1$, $O_2$, $O_3$, and $O_4$, with their conflict relationships expressed in Fig. 2.9, consider the following two different execution orders:
<table>
<thead>
<tr>
<th>OP</th>
<th>$O_2$</th>
<th>$O_3$</th>
<th>$O_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_1$</td>
<td>$\otimes$</td>
<td>$\otimes$</td>
<td>$\otimes$</td>
</tr>
<tr>
<td>$O_2$</td>
<td>$\otimes$</td>
<td>$\otimes$</td>
<td></td>
</tr>
<tr>
<td>$O_3$</td>
<td></td>
<td></td>
<td>$\otimes$</td>
</tr>
</tbody>
</table>

Figure 2.9: The CRT for Example 2.2

**Execution Order 1:** $O_1$, $O_2$, $O_3$, and $O_4$

1. $NCGS_1 = \{\{O_1\}\}$

2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$

3. $NCGS_3 = \{\{O_1\}, \{O_2, O_3\}\}$

4. $NCGS_4 = \{\{O_1, O_4\}, \{O_2, O_3, O_4\}\}$

**Execution Order 2:** $O_1$, $O_2$, $O_4$, and $O_3$

1. $NCGS_1 = \{\{O_1\}\}$

2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$

3. $NCGS_3 = \{\{O_1, O_4\}, \{O_2, O_4\}\}$

4. $NCGS_4 = \{\{O_1, O_4\}, \{O_4, O_3\}, \{O_2, O_4, O_3\}\}$

   $\equiv \{\{O_1, O_4\}, \{O_2, O_4, O_3\}\}$ (by Rule 2)

As shown in Step 4 of Execution Order 2, when $O_3$ is first checked against $\{O_1, O_4\}$, a new group, $\{O_4, O_3\}$, is created since $O_4 \cap O_3$ but $O_1 \cap O_3$; then $O_3$ is checked against
\( \{O_2, O_4\} \), and is added into this existing group (becoming \( \{O_2, O_4, O_3\} \)) since \( O_3 \) is compatible with all operations in this group. However, the new group, \( \{O_4, O_3\} \), is removed since it is a subgroup of \( \{O_2, O_4, O_3\} \) according to Rule 2. In this way, the final \( NCGS_4 \) is the same for two different operation execution orders.

### 2.4 Consistent object identification

For the \textit{MOVIC} algorithm to work, one important parameter has to be provided: the current \( NCGS_{i-1} \), on which the new operation, \( O_i \) is applied to produce \( NCGS_i \). The technical issue here is: how to find the \( CGs \) in the \( NCGS_{i-1} \) for \( O_i \)? Since a \( CG \) in the \( NCGS_{i-1} \) corresponds to an object version made from the original object targeted by \( O_i \), the above issue is converted into the question: how to find the object versions made from the original object targeted by the new operation \( O_i \)? The key to solving this problem is to devise an object identification scheme which is able to identify all object versions made from the same original object.

#### 2.4.1 Requirements for object identification

To work in a multi-version and multi-replica (due to replicated architecture for the stage of shared documents) object-based graphics editing system, the object identification scheme must maintain the following three properties:

1. \textit{Uniqueness}: every object at a site must have an unique identifier.

2. \textit{Traceability}: multiple versions of the same object, \( G \), must have identifiers which can be traced by using the identifier of \( G \).

3. \textit{Consistency}: multiple replicas of the same object at different sites must have the same identifier.
The uniqueness property ensures different objects at a site be distinguishable from each other. The traceability property ensures multiple versions of the same object are traceable by using the identifier of the original object. The consistency property ensures multiple replicas of the same object have the same identifier so that operations applied on one replica be also applied the other replicas. The uniqueness and consistency properties together ensure that an operation targeting an object will be applied to all versions and all replicas of the same object at all sites.

2.4.2 Analysis of object identification issues

Starting from a simple object identification scheme which is able to uniquely identify every object. Let $Id(G)$ denote the identifier of object $G$. Suppose each operation, $O$, has an unique identifier, denoted as $Id(O)$. Then each object can be uniquely identified by the identifier of the operation which created this object. Under this scheme, when object $G$ is created by operation $O$ at a local site, $G$ is assigned a unique identifier which is equal to $Id(O)$, i.e. $Id(G) = Id(O)$. When $O$ is propagated to a remote site, a replica of the same object will be created and assigned the same identifier. When a non-create operation $O$ is applied to an existing object $G$ at the local site, $O$ will take $Id(G)$ as one of its parameters (i.e. $Target(O) = Id(G)$). When $O$ arrives at a remote site, its parameter, $Target(O)$, can be used to find the right replica of the same object to apply. This simple identification scheme works well for single version systems, but fails when multiple versions of the same object can be created due to operation conflicts.

For example, consider three operations, $O_1$, $O_2$, and $O_3$, targeting the same object

\footnote{One way of making the $Id$ for an operation, $O$, is to use a pair $(sid, lc)$, where $sid$ is the identifier of the site at which $O$ is generated, and $lc$ is the sum of the state vector value associated with $O$}
Suppose their conflict relationships are: $O_1 \otimes O_2$, $O_1 \oplus O_3$, and $O_2 \oplus O_3$. Assume these three operations are executed at a site in the order of $O_1$, $O_2$, and $O_3$. To execute $O_1$, $G$ can be found by its original identifier, $Id(G) (= Target(O_1))$. To execute $O_2$, $G$ can still be found by $Id(G) (= Target(O_2))$ because the previous execution of $O_1$ does not change the identifier of $G$. However, after executing both $O_1$ and $O_2$, two versions, $G\{O_1\}$ and $G\{O_2\}$, have been made from $G$ and the original $G$ disappears. When $O_3$ arrives with $Target(O_3) = Id(G)$, both $G\{O_1\}$ and $G\{O_2\}$ must be found in order to combine $O_3$’s effect with them. The question is: how should $G\{O_1\}$ and $G\{O_2\}$ be identified so that they can be traced by using identifier $Id(G)$?

To address the multiple versions identification problem, the simple identification scheme can be extended:

1. to let both versions inherit the identifier of the original object so that they are traceable by using $Id(G)$; and

2. to let one version to include one additional identifier of the operation which triggers the creation of that new version so that multiple versions are distinguishable from each other.

For example, since $O_2$ triggers the creation of a new version, $G\{O_1\}$ could simply take the identifier of $G$, i.e., $Id(G\{O_1\}) = Id(G)$, but $G\{O_2\}$ will take $Id(G)$ plus $Id(O_2)$ as its identifier, i.e. $Id(G\{O_2\}) = Id(G) + Id(O_2)$ (the precise meaning of ‘+’ will be explained at end of this subsection). Clearly, $Id(G\{O_1\}) \neq Id(G\{O_2\})$, and both $G\{O_1\}$ and $G\{O_2\}$ are traceable by using $Id(G)$ since $Id(G)$ is included in both $Id(G\{O_1\})$ and $Id(G\{O_2\})$.

The above extended identification scheme is able to ensure multiple versions of the same object be distinguishable from each other and traceable from the identifier of
the original object. However, it is not able to ensure the consistency of the identifiers of multiple versions of the same object. To illustrate this problem, assume the two conflicting operations in the previous example are executed at a different site in a different order: $O_2$ followed by $O_1$. In this scenario, it will be $O_1$ which triggers the creation of a new version, so $G\{O_1\}$ will take $Id(G)$ plus $Id(O_1)$ as its identifier, i.e. $Id(G\{O_1\}) = Id(G) + Id(O_1)$, but $G\{O_2\}$ will simply take the identifier of $G$, i.e. $Id(G\{O_2\}) = Id(G)$. Clearly, the two replicas of the same object, $G\{O_2\}$, have been identified differently when the two conflicting operations are executed in different orders.

To solve this problem, the previous identification scheme is revised to let both versions include one additional identifier of the corresponding conflicting operation. For the previous example, $G\{O_1\}$ should take $Id(G)$ plus $Id(O_1)$ as its identifier, i.e. $Id(G\{O_1\}) = Id(G) + Id(O_1)$; and $G\{O_2\}$ should take $Id(G)$ plus $Id(O_2)$ as its identifier, i.e. $Id(G\{O_2\}) = Id(G) + Id(O_2)$. With this revised scheme, no matter in which order conflicting operations are executed, multiple replicas of the same object version will be identified consistently.

The object identification scheme would not be completely correct if the following more subtle inconsistency scenario was not discovered and resolved. Given three operations: $O_1$, $O_2$, and $O_3$, targeting the same object $G$. Suppose their conflict relationships are: $O_1 \odot O_2$, $O_1 \odot O_3$, and $O_2 \odot O_3$. First, consider the outcome of executing these operations in the order of $O_1$, $O_2$ and $O_3$. After executing $O_1$, $G$ becomes $G\{O_1\}$, but $Id(G\{O_1\}) = Id(G)$. After executing $O_2$, a new version $G\{O_2\}$ is created and is identified by $Id(G\{O_2\}) = Id(G) + Id(O_2)$. Meanwhile, another version, $G\{O_1\}$, is identified by $Id(G\{O_1\}) = Id(G) + Id(O_1)$ according to
the revised identification scheme. Finally, when $O_3$ arrives, it will be applied to the existing versions, $G\{O_2\}$, directly since $O_3 \odot O_2$. The final outcome of executing the three operations will be two versions: $G\{O_1\}$ with an identifier of $Id(G) + Id(O_1)$, and $G\{O_2, O_3\}$ with an identifier of $Id(G) + Id(O_2)$.

However, if the three operations are executed at a different site in a different order: $O_1$, $O_3$ and $O_2$, the final outcome of executing the three operations will also be two versions: $G\{O_1\}$ with an identifier of $Id(G) + Id(O_1)$, and $G\{O_3, O_2\}$ with an identifier of $Id(G) + Id(O_3)$ because $O_3$ triggers the creation of $G\{O_3\}$. Clearly, the two replicas of the same object version, $G\{O_2, O_3\}$, are identified by two different identifiers (i.e. $Id(G) + Id(O_3)$, and $Id(G) + Id(O_2)$)!

In recognizing this problem, the previous object identification scheme is further revised to let a version’s identifier include the identifiers of all operations (e.g. both $O_2$ and $O_3$) which are conflicting with another operation (e.g. $O_1$), regardless which operation triggers the creation of this new version. Furthermore, it becomes clear that the collection of operation identifiers in the object identifier should be treated as a set, rather than a list. In a set representation, the order of adding a conflicting operation identifier into the object identifier is not significant. For the previous example, we have $Id(G\{O_2, O_3\}) = Id(G) + \{Id(O_2), Id(O_3)\}$, and $Id(G\{O_3\}) = Id(G) + \{Id(O_3), Id(O_2)\}$. Apparently, $Id(G\{O_2\}) = Id(G\{O_3\})$ according to the set theory.

### 2.4.3 The COID scheme

Based on the above analysis, a Consistent Object IDentification (COID) scheme is defined below.
**Definition 2.6.** The COID scheme

The identifier of object $G$ consists of a set of operations identifiers:

$$Id(G) = \{Id(O_1), Id(O_2), \ldots, Id(O_n)\},$$

where $Id(O_i) \in Id(G)$, $1 \leq i \leq n$, iff:

1. $O_i$ is the operation which created $G$, or

2. $O_i$ has been applied to $G$, and $O$ is conflicting with an operation, $O_x$, which has been applied to another version made from $G$. □

In the context of the MOVIC algorithm, the COID scheme can be realized as follows:

1. When operation $O$ creates an original object $G$, $Id(G)$ is constructed as follows:
   $$Id(G) := \{Id(O)\}.$$

2. When operation $O$ triggers the creation of a new version $G'$ from the target object $G$, $Id(G')$ is constructed as follows: $Id(G') := Id(G) + \{Id(O)\}$.

3. When operation $O$ is applied to an existing object $G$, $Id(G)$ is extended to include $Id(O)$ if $O$ is conflicting with any other operation (in $G$ or in any other versions).

4. When operation $O$ is applied to one version of object $G$, every other version of $G$, denoted as $G'$, is checked to see whether $G'$ has the effect of an operation, $O_x$, such that $O_x \otimes O$. If there exists such an $O_x$, and $Id(O_x)$ has not been included in $Id(G')$, then $Id(G')$ is extended as follows: $Id(G') := Id(G') + \{Id(O_x)\}$. 
The COID scheme maintains the *uniqueness* property because the *Create* operation is unique, and any two versions of the same object must have at least one pair of conflicting operations. Moreover, the COID scheme maintains the *consistency* property because for any object, the same set of versions will be replicated at all sites (due to the uniqueness property of the NCGS), and conflict relationships among all operations are the same at all sites. Finally, the COID scheme maintains the *traceability* property because the identifiers of all versions of the same object, $G$, are supersets of $Id(G)$. To answer the question raised at the beginning of this section, the following *Target Object Version Recognition (TOVER)* scheme is defined.

**Definition 2.7.** The *TOVER* scheme

$NCGS_{i-1}$ produced by *TOVER* for operation $O_i$, denoted by $TOVER(O_i)$, is a set of any *CG* corresponding to object $G$ such that $Target(O_i) \subseteq Id(G)$. $\square$

**Example 2.3.**

There are four operations. The first three operations, $O_1$, $O_2$ and $O_3$, are independent of each other. These three operations are targeting the same object $G$, hence, $Target(O_1) = Target(O_2) = Target(O_3) = Id(G)$. Operations $O_1$ and $O_2$ are conflicting with each other. Operation $O_3$ is compatible with both $O_1$ and $O_2$. The conflict relations between these three operations are as shown in Figure 2.11. The last operation, $O_4$, is generated causally after $O_1$ and $O_2$ but independent of $O_3$, as shown in Figure 2.10. Furthermore, $O_4$ is generated after versions have been created (due to the execution of $O_1$ and $O_2$). So $O_4$ is targeting one of the versions, say the one created by $O_2$, so $Target(O_4) = Id(G) + \{Id(O_2)\}$. Lastly, the relationship between
Figure 2.10: Operations’ generation order for Example 2.3

\(O_4\) and \(O_3\) is compatible. Assuming the the initial state of \(G\) is \[\{\}\], the execution of \(O_1, O_2, O_3\) and \(O_4\) is as follows:

1. \(TOVER(O_1) = \{\{\}\}\)
   \[MOVIC(O_1, \{\{\}\}) = \{\{O_1\}\} \text{ where } Id(\{O_1\}) = Id(G)\]

2. \(TOVER(O_2) = \{\{O_1\}\}\)
   \[MOVIC(O_2, \{\{O_1\}\}) = \{\{O_1\}, \{O_2\}\} \text{ where } Id(\{O_1\}) = Id(G) + \{Id(O_1)\}\]
   and \(Id(\{O_2\}) = Id(G) + \{Id(O_2)\}\)

3. \(TOVER(O_3) = \{\{O_1\}, \{O_2\}\}\)
   \[MOVIC(O_3, \{\{O_1\}, \{O_2\}\}) = \{\{O_1, O_3\}, \{O_2, O_3\}\} \text{ where } Id(\{O_1, O_3\}) = Id(G) + \{Id(O_1), Id(O_2)\}\]

4. \(TOVER(O_4) = \{\{O_2, O_3\}\}\)
   \[MOVIC(O_4, \{\{O_2, O_3\}\}) = \{\{O_2, O_3, O_4\}\} \text{ where } Id(\{O_2, O_3, O_4\}) = Id(G) + \{Id(O_2)\}\]
<table>
<thead>
<tr>
<th>OP</th>
<th>$O_2$</th>
<th>$O_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_1$</td>
<td>$\otimes$</td>
<td>$\circ$</td>
</tr>
<tr>
<td>$O_2$</td>
<td>$\circ$</td>
<td>$\circ$</td>
</tr>
</tbody>
</table>

Figure 2.11: The CRT for Example 2.3

In Example 2.3, $TOVER(O_1)$ is first invoked to obtain the initial NCGS for the execution of $O_1$. The set of objects containing only $G\{\}$ will be obtained. Applying $O_1$ to $G\{\}$ does not change the identifier of $G$. In Step 2, $TOVER(O_2)$ will determine that the initial NCGS for $O_2$ is the set of objects containing only $G\{O_1\}$. Applying $O_2$ will result in the versions $G\{O_1\}$ and $G\{O_2\}$. \{$Id(O_1)$\} and \{$Id(O_2)$\} will be added to these versions’ identifiers, so that \(Id(G\{O_1\}) = Id(G) + \{Id(O_1)\}\) and \(Id(G\{O_2\}) = Id(G) + \{Id(O_2)\}\). In Step 3, $TOVER(O_3)$ will determine both $G\{O_1\}$ and $G\{O_2\}$ are in the initial NCGS for $O_3$, since $Target(O_3) \subseteq Id(G\{O_1\})$ and $Target(O_3) \subseteq Id(G\{O_2\})$. $O_3$ will be applied to both versions without any change in their object identifiers. In Step 4, $TOVER(O_4)$ will return the initial NCGS containing only one version \{$G\{O_2, O_3\}$} since $Target(O_4) \subseteq Id(G\{O_2, O_3\})$ but $Target(O_4) \not\subseteq Id(G\{O_1, O_3\})$. Hence, $O_4$ will be applied to produce $G\{O_2, O_3, O_4\}$. The final result of this execution is two versions of $G$: $G\{O_1, O_3\}$ and $G\{O_2, O_3, O_4\}$ with the identifiers of $Id(G) + \{Id(O_1)\}$ and $Id(G) + \{Id(O_2)\}$ respectively.

2.5 Revised conflict definition

In order to reduce the complexity, the discussion of conflict in the previous sections is based on operations targeting the original objects (not versions of objects). Under this assumption, operations may be applied to the same object only if their target object identifiers are equal. However, with the introduction of multiple object versions,
operations may be generated to target specific versions. As the result, operations
targeting the original object and operations targeting versions of that object may be
applied to the same object. These two types of operations have different target object
identifiers. Hence, operations whose target identifiers are not equal may be applied
to the same object. The condition when this occurs between any two operations, $O_a$
and $O_b$, is that:

1. $Target(O_a) \neq Target(O_b)$ and

2. $Target(O_a) \subseteq Target(O_b)$ or $Target(O_b) \subseteq Target(O_a)$.

It is possible that $O_a$ and $O_b$ are independent and $Att.Type(O_a) = Att.Type(O_b)$
and $Att.Val(O_a) \neq Att.Val(O_b)$. Then applying $O_a$ and $O_b$ will result in intention
violation because they will change the same attribute of the same object to different
values. Hence, $O_a$ and $O_b$ should be conflicting operations and should not be applied
to the same object. However, the original definition of conflict (in Definition 2.1)
is not able to detect such conflict because $Target(O_a) \neq Target(O_b)$. Therefore, a
direct conflict relation is introduced (in Definition 2.8) to detect conflicts for any pair
of operations which may be applied to the same object.

**Definition 2.8.** Direct conflict relation

Given two operations, $O_a$ and $O_b$, $O_a$ and $O_b$ have the direct conflicting relation,
denoted by $O_a \otimes_p O_b$, iff:

- $O_a \parallel O_b$

- $Target(O_a) \subseteq Target(O_b)$ or $Target(O_b) \subseteq Target(O_a)$.
• $\text{Att.Type}(O_a) = \text{Att.Type}(O_b)$, and

• $\text{Att.Val}(O_a) \neq \text{Att.Val}(O_b)$. □

**Example 2.3.1**

From Example 2.3, assuming $O_4 \odot D O_3$, the execution of $O_1$, $O_2$, $O_3$, then $O_4$ is as follows:

1. $NCGS_1 = \{\{O_1\}\}$

2. $NCGS_2 = \{\{O_1\}, \{O_3\}\}$

3. $NCGS_3 = \{\{O_1, O_3\}, \{O_2, O_1\}\}$

4. $NCGS_2 = \{\{O_1, O_6\}, \{O_2, O_3\}, \{O_2, O_4\}\}$

In this execution, by using the direct conflict definition, even though $\text{Target}(O_3) \neq \text{Target}(O_4)$ the conflict between $O_3$ and $O_4$ is detected. As a result, versions are made for $O_3$ and $O_4$ and their intentions are preserved.

The initial $NCGS$ for any operation, $O$, produced by $TOVER(O)$ defines the scope of application for $O$. $O$ may be added to any compatible group in $NCGS$ after the execution of $O$. For any two operations, $O_a$ and $O_b$, such that $\text{Target}(O_a) \subset \text{Target}(O_b)$, then the scope of $O_a$ is smaller than the scope of $O_b$ since $NCGS$ for $O_a$ is a subset of $NCGS$ for $O_b$. If $O_a$ and $O_b$ are independent, then they will be executed in different orders at different sites. In certain situations, inconsistencies may occur. This situation is as shown in Example 2.4.
Example 2.4.

There are four operations, $O_1$, $O_2$, $O_3$, and $O_4$, with the same dependency relationships as in Example 2.3. The conflict relationships between $O_1$, $O_2$ and $O_3$ are as shown in Figure 2.12. Again, $Target(O_4) = Id(G) + \{Id(O_2)\}$. This time, assume $O_4$ and $O_3$ are not directly conflicting (i.e. $Att.Type(O_4) \neq Att.Type(O_3)$). These operations may be executed in different orders:

**Execution order 1**: $O_1$, $O_2$, $O_3$, and $O_4$

1. $NCGS_1 = \{\{O_1\}\}$
2. $NCGS_2 = \{\{O_5\}, \{O_2\}\}$
3. $NCGS_3 = \{\{O_1, O_3\}, \{O_1\}\}$
4. $NCGS_4 = \{\{O_5, O_3\}, \{O_2, O_4\}\}$

**Execution order 2**: $O_1$, $O_2$, $O_4$, and $O_3$

1. $NCGS_1 = \{\{O_1\}\}$
2. $NCGS_2 = \{\{O_1\}, \{O_2\}\}$
3. $NCGS_3 = \{\{O_3\}, \{O_2, O_4\}\}$
4. $NCGS_4 = \{\{O_1, O_3\}, \{O_2, O_4\}, \{O_4, O_3\}\}$
In Execution order 1, when applying $O_4$, it was not compared for conflict with $O_3$ because the object $O_3$ was applied to was not in the scope of $O_4$. However, in Execution order 2, when applying $O_3$, $O_3$ and $O_4$ were compared for conflict since the object $O_4$ was applied to is in the scope of $O_3$. As a result, the final effect between these two execution orders is inconsistent.

To produce consistent effect at all sites, no matter if the two operations are compared for conflict or not, the same result should be produced at all sites. There are two choices:

1. Produce the result as if $O_4$ and $O_3$ are compatible as in execution order 2. However, this would cause a problem in execution order 1, because $O_4$ will have to be forced to compare with $O_3$, even though the object $O_3$ is applied to is not in the scope of $O_4$. This is a very complicated task (and may well be impossible).

2. Produce the result as if $O_4$ and $O_3$ are conflicting as in execution order 1. To ensure execution order 2 produces the same result, a new conflict definition is required so that when compared, $O_4$ and $O_3$ are conflicting. With this solution, only a new conflict definition is required and no other change needs to be made to the existing algorithms. Therefore, this solution has been adopted.

This new conflict relation is called *indirect conflict*. Its definition (in Definition 2.9) captures the general situation where this conflict occurs. The cause of indirect conflict is that the two involved operations, $O_a$ and $O_b$, have different target object scopes i.e. $\text{Target}(O_a) \subset \text{Target}(O_b)$. This difference in target object scope is caused by an operation, $O_c$ where $O_c \otimes D O_a$ and $O_b$ targets the version created by $O_c$. 
**Definition 2.9.** Indirect conflict relation

Given two independent operations, $O_a$ and $O_b$, $O_a$ and $O_b$ have the indirect conflicting relation, denoted by $O_a \otimes_I O_b$, iff there is an operation $O_c$ and compatible group $CG$ such that:

- $O_c \otimes_D O_a$,
- $Target(O_b) = Id(CG)$ and $O_c \in CG$, and
- $Target(O_c) \subseteq Target(O_b)$.  

Based on direct and indirect conflicts, the conflicting and compatible relationships are redefined in Definition 2.10. The conflict relations in Example 2.3 and 2.4 are also revised to include $O_4$ as shown in Figure 2.13 and 2.14.

**Definition 2.10.** Compatibility relationship

Given two operations, $O_a$ and $O_b$, $O_a$ and $O_b$ are:

- **conflicting**, denoted by $O_a \otimes O_b$, iff either $O_a \otimes_D O_b$ or $O_a \otimes_I O_b$;
- **compatible**, denoted by $O_a \oplus O_b$, iff $O_a$ does not conflict with $O_b$.  

The object identification relies on the definition of conflict. Due to the changes made to the conflict definition, the object identification scheme also needs to be revised. The identifier of a conflicting operation is added to the object’s identifier because conflict causes the creation of versions. However, with the new conflict definition, indirect conflict operations does not cause version creation. Therefore, in COID scheme, conflict refers only to direct conflict.
2.6 Version creation

How to create the required object versions? There are two general approaches to constructing any version $G'$ from version $G$ given the compatible group, $CG_{\mathcal{C}}$, of $G$:

1. **Construct the version from scratch.** Start with the $Create$ operation that created $G$. $G'$ is created by:

   (a) re-executing this $Create$ operation to create a new object $G'$, and
   (b) applying all operations in $CG_{\mathcal{C}}$ to $G'$.

2. **Construct the version from an existing object** [18]. Start with $G$. $G'$ is created by:

   (a) making an exact duplicate of $G$ as $G'$,
   (b) undoing operations applied to $G'$ which are not in $CG_{\mathcal{C}}$, and
   (c) applying operations in $CG_{\mathcal{C}}$ which have not been applied to $G'$.
Both approaches can produce the correct version, however, only one approach is feasible. Constructing the version from scratch may not be feasible, because this approach requires all operations that should be applied to this version to be kept by the system. For to this reason, versions in GRACE are constructed from existing objects.

In order to illustrate how versions are constructed from existing objects, consider the scenario with three conflicting operations, $O_1$, $O_2$ and $O_3$. Assuming these three operations are targeting the same object, $G$, with the initial state of $\{\}$. $O_1$ is first executed resulting in $G\{O_1\}$. When executing $O_2$, versions $G_1\{O_1\}$ and $G_2\{O_2\}$ need to be made. Since $G$ has the same compatible group as $G_1$, therefore, $G$ can simply be converted to $G_1$ by changing its identifier (i.e. $Id(G_1) = Id(G) + Id(O_1)$). $G_2$ can be constructed from $G_1$ by duplicating $G_1$, then undoing $O_1$ from $G_2$ and applying $O_2$ to $G_2$. The identifier of $G_2$ also needs to be changed by removing $Id(O_1)$ and adding $Id(O_2)$ to its object identifier. When executing $O_3$, version $G_3\{O_3\}$ will need to be constructed. $G_3$ is constructed in the same method as $G_2$. The object that $G_3$ is to be constructed from can be either $G_1$ or $G_2$. Say $G_2$ is used to construct $G_3$. $G_2$ is first duplicated to produce $G_3$. Then $O_2$ is undone from $G_3$ and $O_3$ is applied to $G_3$. Finally, the identifier of $G_3$ is changed by removing $Id(O_2)$ and adding $Id(O_3)$ to its object identifier.

2.7 Convergence

2.7.1 Intra-object convergence

In addition to resolving intention violation, the MOVIC algorithm also solves intra-object divergence caused by conflicting operations (as described in Section 1.3.2). For
example, there are two conflicting \textit{Move} operations, $O_1$ and $O_2$, to move the same object to positions $x$ and $y$ respectively. Executing these two operations in any order will result in versions $G_1$ in position $x$ and $G_2$ in position $y$ being created at all sites. Since the multiple object version scheme ensures the effect of all conflicting operations are preserved, convergence is also ensured. Therefore, serialization is not required to solve intra-object divergence problems caused by conflicting update operation. None-conflict or compatible operations are commutative, and can be applied in any order without causing divergence. For example, a \textit{Fill} and a \textit{Move} operation targeting the same object can be applied in any order. The end result is that the object is moved and color changed. Hence, update operations can be applied in any order and convergence is ensured.

### 2.7.2 Inter-object convergence

The \textit{MOVIC} algorithm solves intra-object divergence, but it does not address the problem of inter-object divergence. Two overlapping objects can still be created concurrently, and their overlapping order will be different depending on their execution order. So serialization is still required for \textit{Create} operations. Update operations do not require serialization, therefore, serialization need only be applied amongst \textit{Create} operations.

Another inter-object divergence problem occurs when versions made from the same object overlap each other. How to ensure inter-version convergence where the same overlapping order between versions appears at all sites? One approach is to treat conflicting operations which cause the creation of new versions, like \textit{Create} operations, and produce a serialized overlapping effect amongst versions. For example, versions $G_1$ and $G_2$ are made for the two conflicting operations $O_1$ and $O_2$ respectively.
Assuming $O_1 \Rightarrow O_2$, then all sites should produce the effect of $O_2$ on top of $O_1$.

However, there are some differences between $Create$ operations and conflicting operations which cause the creation of versions. An object can only be created by one $Create$ operation. The creation of a version may be caused by one of many conflicting operations. For example, there are three independent operations with the compatibility relationships of $O_1 \circ O_2$, $O_1 \circ O_3$ and $O_2 \circ O_3$. The execution of these three operations would result in $O_1$ being applied to version $G_1$ and $O_2$ and $O_3$ being applied to the version $G_2$. This means $G_2$ can be created by either $O_2$ or $O_3$, depending on which operation is executed first. It is possible that the serialization order between these three operations is: $O_2 \Rightarrow O_1 \Rightarrow O_3$. So if $O_2$ caused the creation of $G_2$, then $G_2$ is below $G_1$. If $O_3$ caused the creation of $G_2$, then $G_2$ is above $G_1$. Apparently, inter-object divergence will occur under this situation.

An approach that will maintain convergence under such situation is by serializing all conflicting operations (instead of serializing only the conflicting operations that caused the creation of versions). In the above example, the serialization order is $O_2 \Rightarrow O_1 \Rightarrow O_3$. Since $O_2 \Rightarrow O_3$, $O_2$ will cause the creation of $G_2$, thus $G_2$ is below $G_1$. Since this is the only valid serialization effect, convergence is ensured. However, this approach has a side effect. If these three operations arrive at a site in the order of $O_3$, $O_1$ followed by $O_2$, then the following execution sequence would be produced:

1. After executing $O_3$ and $O_1$, $O_3$ causes the creation of $G_2$ (and $O_1$ causes the creation of $G_1$), so $G_2$ is above $G_1$.

2. When $O_2$ arrives, undo/do/redo is trigged so now $O_2$ causes the creation of $G_2$, so $G_2$ is now below $G_1$.

This effect can cause confusion to the users because $G_2$'s layering position is changed
from above to below, even though no user generated any operation to change the layering of $G_2$ or $G_1$. Unfortunately, so far it has not been possible to identify a solution that will fix this problem. However, it should be stressed that this side effect only causes divergence during the intermediate execution effect and only under some rare situations. The final effect, when all independent operations have been executed, remains convergent. Therefore, any solution to this problem should not cause significant deterioration to the system’s performance.

Having addressed the layer ordering convergence issue between versions, there is another question: what is the layer ordering between these versions and other objects in the document? Since versions are made to represent an object, these versions should have the layering order of this object. To be more precise, if an object, $G$, is between objects $G_x$ and $G_y$, then versions made from $G$ should also be between $G_x$ and $G_y$, as defined in Definition 2.11. Maintaining this order is a trivial implementation issue.

**Definition 2.11.** Version layering property (VLP)

Given any three objects, $G$, $G_x$ and $G_y$, if $G$ is above $G_x$ and below $G_y$, then immediately after conflicting operations are applied to $G$, versions of $G$ should also be above $G_x$ and below $G_y$. □

### 2.8 Comparison within the graphics editing domain

Most existing collaborative graphics editing systems have adopted a conflict-prevention approach based on locking. Example systems based on locking include: Aspects [113],
Ensemble [76], GroupDraw [43], and GroupGraphics [84]. In these systems, a user has to place a lock on an object before editing it, thus preventing other users from generating conflicting operations on the same object. For locking to work, however, there must be a coordinating process in the system which keeps track of which object(s) have been locked so it can grant/deny permissions for locking requests. The problem with locking is that when an editing operation is generated, it has to wait for at least the round-trip time of sending a request message to the coordinating process and receiving a grant message back, before it can be executed (if it is allowed) at the local site. This round-trip delay in the Internet environment may significantly degrade the system’s responsiveness. Various techniques have been proposed to overcome this problem. For example, Ensemble allows conflict-free operations to execute immediately without waiting for approval. While in GroupDraw, locally generated operations are executed immediately and a message is sent to the coordinating process. If the coordinating process does not approve of the operation, then the effect of that operation is undone, which may cause abnormal phenomena at the user interface.

In contrast to conflict-prevention approaches like locking, the multiple object version strategy proposed in this chapter allows conflict to occur. It provides mechanisms (multiple object versions) to accommodate conflicting operations in a consistent way. The conflict resolution is left to the users of the system. The major advantages of this approach are that it helps to achieve high responsiveness in the system and preserves the work concurrently produced by multiple users in the face of conflicts. However, locking does have the merit of reducing conflicts by enforcing mutual exclusion. In fact, locking has been found to be complementary and compatible with the optimistic concurrency control strategies; and proposed is a novel optional locking scheme (in
contrast to existing *compulsory* locking schemes) to enhance the consistency maintenance capability of the system [107]. The investigation into the integration of this optional locking scheme with multi-version approach in the graphics editing domain is reported in Chapter 4.

Another alternative conflict-resolution approach is *serialization*. With this approach, operations can be executed as soon as they are generated to give a quick response. Before an operation is executed, it must be checked against previously executed operations for possible conflict. If a conflict is detected, a total ordering (i.e., serialization) between operations is used to determine which operation's effect will appear. Examples of such systems are: GroupDesign [55] and LICRA [53]. This approach is essentially the *single operation effect* (determined by a total ordering) approach discussed in Section 2.1. For problems with this approach and its major differences with the *all operations effect* approach, refer to the analysis in Section 2.1.

The most closely related work is the Tivoli whiteboard meeting-supporting tool developed at Xerox PARC [71]. Tivoli also use multiple object versions (called 'replicas' in Tivoli) to accommodate the effects of conflicting operations. The major difference between the Tivoli approach and the GRACE approach is that, in Tivoli, a conflict occurs whenever two concurrent operations target the same object, whereas in GRACE, a conflict occurs only when two concurrent operations target the same object and change the same attribute to different values. Tivoli use a very simple algorithm for controlling the creation of the multiple copies of an object: whenever two concurrent operations target the same object, two copies of the object are created. Consequently, Tivoli does not allow compatible operations (according to our
definition) to be applied to the same object (e.g. concurrent *Move* and *Fill* operations cannot be applied to the same object), resulting in unnecessary object versions. Another limitation with Tivoli is that it is able to support only two collaborating sites in a session [71]. In contrast, GRACE is designed to support an arbitrary number of sites in a collaborative editing session. As far as it is known, GRACE is the only system which is able to minimize the number of object versions for accommodating the combined effects of conflicting and compatible operations.

2.9 Comparison between text and graphics domain

GRACE and REDUCE are collaborative editing systems based on different document types, text for REDUCE and object graphics for GRACE. However, both REDUCE and GRACE systems share the the same inconsistency problems of divergence, causality violation, and intention violation. For both types of systems, divergence can be solved by serialization, and causality violation can be solved by enforcing causal order execution. However, there is no common method that can solve intention violation for both systems, i.e. operational transformation is used for text systems and multiple object versions are used for object graphics systems (as shown in Figure 2.15). Why is that? In order to understand their differences, a comparison is made between the document structure and operations in REDUCE and GRACE. This comparison is then analyzed and the results presented.

2.9.1 Comparison

Document structure

How effects are preserved is dependent on the document’s data structure. In REDUCE, the text document is represented by a sequence (list) of plain characters.
<table>
<thead>
<tr>
<th></th>
<th>REDUCE</th>
<th>GRACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Causality Preservation</td>
<td>Enforce causal order execution</td>
<td>Enforce causal order execution</td>
</tr>
<tr>
<td>Convergence</td>
<td>Serialization or Operational transformation</td>
<td>Serialization or Multiple object version</td>
</tr>
<tr>
<td>Intention Preservation</td>
<td>Operational transformation</td>
<td>Multiple object version</td>
</tr>
</tbody>
</table>

Figure 2.15: *The comparison of concurrency control schemes in REDUCE and GRACE*

The effect of inserting characters into the document is accommodated by shifting the characters to the right of the inserting position towards the end of the document to make space for the newly inserted characters. This results in an increase in the overall document length. Delete is simply the opposite of insert. The effect of deleting a character is produced by removing that character and shifting all the characters on the right of the deleting position towards the left. This will result in decreasing the overall document length.

A GRACE document is represented by a list of objects (object list). The position in the list determines the layering of the objects. The object at the start/left side of the list is on top of all other objects. Adding/deleting an object is similar to inserting/deleting a new character in the text document.

If we simply regard each character in REDUCE as an object (character object), then the document structures between REDUCE and GRACE are the same. Both document structures are simply a list of objects (or linear data structure). The operations used to edit this document can be classified into operations for adding objects, removing objects, and updating objects.
Adding/removing objects

The operation for adding an object is Insert for REDUCE, and Create for GRACE. What is the difference between these two types of operation?

The intended (syntactical) effect of insert is to place the string between any two consecutive characters. However, the positions of these two characters may be shifted due to other concurrent insert or delete operations. For example, assuming the shared document initially contains the sequence of characters "ABCDE" ("A" is in position 0 and "B" is in position 1, etc.). Let $O_1$ and $O_2$ be two concurrent insert operations. Suppose $O_1 = \text{Insert}["12", 1]$ which intends to insert string "12" at position 1, i.e. between "A" and "BCDE"; and $O_2 = \text{Insert}["34", 2]$ which intends to insert string "34" at position 2, i.e. between "AB" and "CDE". After their execution, the intention-preserved result should be "A12B34CDE". However, the execution of these two operations in the order of $O_1$ followed by $O_2$ will produce the result of "A1342BCDE", which violates the intention preservation property. This result is produced due to the execution of $O_1$ changing the inserting position of $O_2$. As a result, the execution of $O_2$ will cause "34" to be inserted in the wrong position. To solve this problem, operational transformation is used. By using operational transformation, $O_2$ will be transformed against $O_1$ so $O_2$ will take into account the effect of $O_1$ on the document. Hence, the inserting position of $O_2$ will be corrected to become $O_2 = \text{Insert}["34", 4]$, so that when $O_2$ is applied to "A12BCDE" it will produce "A12B34CDE".

Create operations add new objects to the document. The newly created objects are on top of all existing objects. Create and Insert are similar, in that a string/object is added to the document. However, the difference lies in where
the string/object is inserted/created within the document. The effect of *Create* is preserved if the object created is on top of all existing objects (this is in contrast to *Insert*, where the insertion has to be between two characters). The effect of *Create* operations can be preserved by simply adding the newly created objects to the start/left of the object list as shown in Figure 2.16. This will ensure the newly created object is on top of all existing objects because all existing objects must be in the list. Hence, the effect of insert is preserved. However, overlapping divergence may occur amongst objects created by concurrent operations as shown in Figure 2.16. Operational transformation cannot be used to transform *Create* operations because *Create* operations do not explicitly specify the positions where the objects are to be created. Therefore, this divergence problem is solved by serialization as discussed in Section 2.7.2.

The operation for removing an object is *Delete* for REDUCE, and *Destroy* for GRACE. What is the difference between these two types of operation?
Delete operations suffer similar problems as Insert where the characters to be deleted may be shifted due to the execution of concurrent operations. As a result, incorrect characters are deleted. This intention violation problem can also be solved by operational transformation.

One may suspect that Destroy operations will suffer the same intention violation problem as Delete. This is because the object to be destroyed can be anywhere in the document and its position may be changed by concurrent Create and Destroy operations. This would be true if Destroy operations specify their target object by the position in the object list [20]. However, in GRACE, Destroy operations specify their target objects by the objects' unique identifiers. Object identifiers will never be changed. Therefore, by using identifiers to determine the target object, the correct object will always be found. Therefore, operational transformation is not needed.

Updating objects

Update operations do not add or remove any object to/from the document. Instead, they change/replace the attribute values of their targeted objects. Currently, REDUCE does not support any operation for changing objects, since it only supports plain text characters (without any attributes).

GRACE supports update operations such as Move, Fill and Resize. As with Destroy, the target objects for update operations also have to be determined. This also is achieved by using the object identifiers to find the objects. After the target objects have been found, update operations are applied to change their attribute values. However, intention violation may occur when two or more conflicting operations are applied to the same object. Therefore, multiple object versions are produced to resolve conflicts.
Text editing systems, whose characters contain attributes such as font type and font size, etc., may have update operations to change characters’ attributes. If so, conflicts may occur on those operations. For example, two operations are concurrently generated to change the font size of the same character(s) to different values. The multiple object version approach may be also used to resolve this conflict. In this case, multiple versions of the same character(s) will be created, one version to satisfy the intention of each conflicting operation.

2.9.2 Generalization

Object targeting systems

From the analysis in the above section, it is obvious that there are two different methods (which have been used) to find objects to apply operations to. The first method is to find objects using their position in the document. This method will be called targeting by position. The requirements for this method is that all positions have a linear order and each object occupies a different position. Operations are generated with the positions of their target objects. These positions, specified by the operations, are used to find their target objects at the remote sites.

A major problem with targeting by position is that the positions of objects may change. The change in position of an operation’s target object, caused by the execution of concurrent operations, may result in the wrong object being selected (as described in Section 2.9.1). Therefore, operational transformation needs to be used to ensure the correct object is selected every time.

The second method to find objects is by using the unique identifier of each object. This method will be called targeting by identifier. The requirement for this method is that each object must have a unique identifier. Operations are generated with the
identifier of their target object. These identifiers, specified by the operations, are used to find their target objects at the remote sites.

The difficulty of targeting by identifier is to be able to generate unique object identifiers concurrently from different sites. The object identification scheme presented in Section 2.4 is designed to solve this problem. Targeting by identifier does not suffer from the same problem as targeting by position, where the position of the target operation may be changed by the execution of concurrent operations. This is because object identifiers can never be changed by the execution of any operation.

These two targeting systems have their relative advantages and disadvantages. Targeting by position has the advantage that the target object can be found at no cost (since the location is specified). However, it has the disadvantage of the time taken to transform operations and the complexity of implementing operational transformation. Targeting by identifier has the advantage of not requiring operational transformation. However, it has the disadvantage of requiring memory space to store the identifiers. Furthermore, it takes time to search through the collection of objects to find the target object.

Generally speaking, both targeting systems can be used in collaborative text or graphics editing systems. However, due to the different nature of the documents, targeting by position is commonly used for collaborative text editing systems [31, 90, 105], while targeting by identifier is commonly used for collaborative graphics editing systems [71, 102].

For text documents, the order between characters is important (there must be a linear order). All sites have to maintain the same character order, so targeting by position is suitable. If targeting by identifier is to be used, this character order
still needs to be maintained, and in addition, each character may potentially need an identifier. This is inefficient because the space required to store the identifier is relatively large compared to the size of the character object. This approach may be unfeasible when the document is large.

For graphics documents, although the order between the objects is used to determine the layering order in GRACE, this order is generally not essential, since most objects do not overlap. Consistent overlapping can be maintained without imposing a strict order on all objects. This means objects may be stored in different orders at different sites. If so, targeting by position is not suitable. However, if one must use targeting by position, object ordering can be enforced at all sites, then targeting by position may be used. Targeting by identifier works without enforcing object ordering. Furthermore, the space occupied by the identifiers is very small compared to the size of graphical objects. On average, the number of graphical objects in a document is significantly less than the number of characters, so the time for searching objects is also significantly less.

In summary, there are two targeting systems that may be used to find target objects. These targeting systems use different concurrency control techniques to ensure the correct target objects will be found. These systems have different advantages and disadvantages. Their suitability for a particular type of collaborative editing system depends on the nature of the document, such as the ordering between objects, and the size and the number of objects.
Concurrency control algorithms

It should be now apparent that operational transformation and the multiple object version scheme solve inconsistency problems of different natures. Operational transformation is used for finding the correct target object when targeting by position is used. Whereas the multiple object version scheme is used for resolving conflict caused by concurrent operations changing the same attribute of the same object to different values. The multiple object version scheme is independent of the targeting system and it can be used regardless of which targeting system is in place.

Operational transformation and the multiple object version scheme are not competing techniques. They are complimentary techniques used to solved different problems in collaborative editing systems. For a collaborative editing system that uses targeting by position and contains objects with at least one attribute, then both operational transformation and the multiple object version schemes can be used to maintain consistency.

2.10 Conclusions

In this chapter, a novel multiple object version approach to conflict resolution in real-time collaborative graphics editing systems has been proposed. This approach is able to preserve the work concurrently produced by multiple users in the face of conflicts, and to minimize the number of object versions for accommodating combined effects of conflicting and compatible operations.

Firstly, conflict and compatible relationships among graphics editing operations were analyzed and defined. Then a formal specification of a unique combined effect
for any group of conflicting and compatible operations was proposed in three Combined Effect Rules. It was found that a unique combined effect complying with the three Combined Effect Rules can be derived from the inherent conflict relationships among any group of operations targeting the same object. To capture the conflict relationships among any group of operations, the conflict relation matrix and triangle, the compatible group set, the rules for transforming one compatible groups set to another equivalent set, and the Normalized Compatible Group Set (NCGS) were defined. Moreover, the uniqueness of the NCGS was proved, of the combined effect based on the NCGS was specified, and that this combined effect complied with the three Combined Effect Rules was verified.

Another major technical contribution of note in this chapter is the design of a distributed algorithm, MOVIC, which is able to incrementally construct the NCGS for any group of operations regardless of the order in which these operations are executed. To support the MOVIC algorithm, an object identification scheme COID which is able to uniquely and consistency identify all objects, and to trace all object versions made from the same original object, has been described. In addition, the TOVER scheme has been devised to determine the correct object versions belonging to the application scope of any operation. These objects are determined based on the identifier of the target target object of this operation and the identifiers of existing objects at a remote site.

Conflict relationships among operations targeting different original objects but with overlapping application scopes, have been studied. The concepts of direct conflict and indirect conflict have been introduced to solve inconsistency problems caused by these special conflict relationships.
In addition to intention preservation, the multiple object version scheme also solves intra-object divergence problems. This scheme maintains intra-object convergence without requiring the system-initiated undo/do/redo of executed operations. Furthermore, a conflict layering scheme based on serialization has been devised to preserve inter-object convergence. This scheme ensures all objects appear in the same layering order at all sites, in the presence of concurrent and conflict operations.

A comparison between this work and related work in the domain of graphics editing has shown that this work is the first scheme based on attribute level conflict. This scheme has an advantage over object level conflict because of the small granularity of conflict, hence conflict is less likely to occur. Then a comparison was made between text and graphics collaborative editing domains. This comparison discovered that the text and graphics editing systems use different methods to find objects for applying operations to. Furthermore, operational transformation (used in text domain) and multiple object version scheme are complimentary concurrency control techniques which solve inconsistency problems of different natures.

The multiple object version approach lays the foundation for future research in collaborative graphics editing systems. This by itself is not a complete solution to resolving conflicts in collaborative editing systems. Other complementary techniques should be integrated to work in conjunction with the multiple object version technique. Two techniques to work in conjunction the multiple object version technique have been devised. In the next chapter, an Any Undo scheme is proposed which allows users to undo any of the previously executed operations, including conflicting operations. In Chapter 4, an optional locking scheme is proposed to help minimize the chance of conflict operation generation.
Chapter 3

Maintaining consistent undo effect of any operation

The previous chapter deals with consistency maintenance in the execution of user generated operations. This chapter also deals with consistency maintenance, but in the context of user initiated undo of previously executed operations. Furthermore, consistency needs to be maintained under the condition that users may undo any executed operation in any order. The undo effect of any operation should reverse its execution effect (including multiple object version effect). Hence, the work presented in this chapter is a natural extension of the results presented in Chapter 2.

3.1 Any Undo

Undo is a useful and widely supported feature in single-user interactive applications. Undo can be used to recover from erroneous operations, learn new system features by try-and-failure, and explore alternative solutions by backtracking [37, 112, 117]. Undo is especially important for graphics editing systems because a major mistake can be easily made with a single mouse click [4].

Undo in multi-user collaborative applications is particularly valuable because:
1. multi-user applications should have all the features found in single-user applications;

2. the potential cost of an individual user’s mistake is multiplied many times since it may adversely affect the work of a large number of collaborative users; and

3. the number of alternatives to be explored in a collaborative setting increases due to the presence of many users [24].

*Any Undo* [99] is an undo solution which allows users to undo any operation at any time with guaranteed success. Any Undo can be used to build different undo models such as single-step undo, chronological undo, and selective undo [85]. This provides the flexibility of allowing users at different sites to use different undo models. Each user can choose a favorite undo model independent of other users. Furthermore, Any Undo allows users to undo only locally generated operations (local undo), or operations generated by all users (global undo). Hence, Any Undo is particularly suitable for collaborative editing systems.

Currently, there are a number of solutions for undo in collaborative text editing systems [85, 89, 99]. In contrast, the work on undo solutions for collaborative graphics editing systems are very limited. Furthermore, several researchers have indicated that designing an undo solution for this type of system is a challenging task [9, 42, 71]. This chapter focuses on Any Undo for GRACE.

The rest of this chapter is organized as follows. Section 3.3 examines what needs to be done for undo/redo. Section 3.4 discusses how to produce the undo/redo effect on objects. Section 3.5 discusses how to remove and recreate object versions. Section 3.7 combines the results of the two previous sections to produce the undo and redo
algorithms. Section 3.8 discusses operation dependent issues in effect generation. Section 3.9 compares this work with related work. Finally, the conclusion is presented in Section 3.10.

3.2 Generic Any Undo effect

Any Undo was proposed by Sun in [99] and the techniques were developed for REDUCE. Most techniques and algorithms used in REDUCE cannot be used in GRACE. However, the specification of an Any Undo effect is independent of the semantics of the operation and is applicable to any type of editing system. This Any Undo effect has been adopted for GRACE.

Let $DS_0$ be the initial document state, $O_i$ be the $i$th operation executed on the document, $Undo(O_i)$ be the command to undo operation $O_i$, and $DS_n$ be the document state after applying a sequence of operations from $O_1$ to $O_n$ on $DS_0$, denoted as: $DS_n = S(O_1 \circ ... \circ O_{i-1} \circ O_i \circ O_{i+1} \circ O_n)$.

**Definition 3.1.** Undo effect

After executing $Undo(O_i)$ in document state $DS_n$, the following undo effects should be achieved:

1. the effect of $O_i$ is eliminated;

2. the original effects of operations from $O_1$ to $O_{i-1}$ are retained; and

3. the effects of operations from $O_{i+1}$ to $O_n$ are retained as if they were executed without $O_i$. 

□
Definition 3.2. Undo property

Given the current document state $DS_n$, undoing all operations between $O_i$ and $O_n$ in any order must restore the document to state $DS_{i-1}$.  

This property is essential for any undo solution to be an effective tool for supporting error recovery and alternative exploration because it ensures that the document can be restored to any previous document state by undoing all operations executed after that state, regardless of the order in which these operations are undone. In particular, by undoing all operations in a session, the document can always be restored to its initial state.

It can be shown that if undoing an arbitrary operation in $DS_n$ achieves the undo effect as defined in Definition 3.1, then undoing the group of operations between $O_i$ and $O_n$ will surely maintain the undo property. This is because undoing operations between $O_i$ and $O_n$ will eliminate the effects of these operations but retain the original effects of operations between $O_1$ and $O_{i-1}$, which are the operations leading to the document state $DS_{i-1}$. Therefore, the document state must be in $DS_{i-1}$ after undoing all operations between $O_i$ and $O_n$ in any order.

3.3 Issues in graphics DO, UNDO, and REDO

What is the meaning of Definitions 3.1 and 3.2 in the context of object graphics editing systems? What effect should be produced when undoing an object graphics editing operation that will satisfy these two definitions? The purpose of undo is to inverse the effect of the normal execution of that operation or DO. Therefore, to be able to answer these questions, one has to understand the DO effect of object graphics
operations. Understanding the DO effect is essential for determining the undo effect, and conversely, knowing the undo effect is essential for determining the redo effect. This section examines what DO does, and what issues undo/redo should deal with.

In DO, immediately after an operation is applied to an object, its effect will be visible on that object, which means the attribute takes on the value specified by this operation. This remains so until another operation is executed to change the same attribute. When this happens, the effect of the first operation is overwritten by the second. Once an operation has been overwritten, its effect is not visible. For example, $O_1$ and $O_2$ are Fill operations to change the color of an object $G$ to red and blue respectively. The execution of operation $O_1$ charges the color of object $G$ to red. Hence the effect of $O_1$ is visible. Then the execution of operation $O_2$ changes the color of $G$ to blue. So the effect of $O_1$, red, has become not visible because it has been overwritten by the effect of $O_2$, blue.

The above is applicable only to non-concurrent operations. Concurrent operations targeting the same object and changing the same attribute to different values are regarded as conflicting operations. Conflicting operations will be applied to different versions by the multiple object version scheme. For example, if $O_1$ and $O_2$ are conflicting, then their execution will result in two versions of $G$, $G_1$ with the effect of $O_1$, red, and $G_2$ with the effect of $O_2$, blue.

From the above analysis, a generalization can be made regarding the task of DO:

1. If necessary, create a version or versions for the operation to apply to.

2. Make the effect of the operation visible.

Undo should inverse the effect of DO. This is simple if the operation selected for undo is always the last operation executed. The last operation can always be undone.
However, with Any Undo, any operation may be selected for undo at any time. How to ensure every operation is undoable? The effect of the operation to be undone may be either visible or not visible on an object, how to deal with these situations?

Undo also has to deal with multiple object versions created during DO. Undoing a conflict operation should result in the removal of multiple versions created during DO. How to remove versions? Moreover, how to preserve the effects of operations applied to versions that are to be removed due to undo?

Redo is similar to DO. However, redo is more complicated than DO because any undone operation can be selected for redo in any order. After DOing (executing) an operation, its effect is always visible on the object it is applied to (until another operation is applied to overwrite its effect). This is not always the case for redo. Redo should inverse the effect of undo. When an operations is undone, its effect may be visible or not visible. Therefore, after redoing an operation, its effect may also be visible or not visible.

Redo also has to deal with multiple object versions removed during undo. Hence, instead of removing versions, redo may need to recreate versions. This also has to be done under the condition that operations may be executed in any order. This may result in an operation needing to be redone on a version that was removed. How to preserve the redo effect in such circumstances?

In summary, the solution to both undo and redo needs to deal with two separate issues:

1. generic undo/redo issues of producing the visible undo/redo effect of an operation on a single object; and

2. multiple version specific issues of producing the undo/redo effect on multiple
versions.

The solution to these problems will be examined in two separate segments (Sections 3.4 and 3.5) then the ensuing results will be integrated to form the schemes for undo and redo (in Section 3.7).

3.4 Undo effect on objects

This section focuses on producing the undo/redo effects of operations on individual objects. The section is divided into three subsections. The first subsection presents an undo/redo effect for object graphics editing systems. The second subsection describes how to generate an operation, which when applied to an object, will produce the correct undo/redo effect. The third subsection discusses how to represent operations after they have been undone/redo. The techniques provided in this section are generic and can be used by any object graphics editing system.

3.4.1 Undo/redo effect

The overwritten effect in object graphics editing systems is essential in determining the DO effect, and hence the undo/redo effect. In GRACE, independent operations targeting the same object and changing the same attribute values do not overwrite each other. This is because they either change the same attribute to the same value (i.e. the execution of the second operation does not change the attribute value set by the first operation), or they change the same attribute to different values which will result in conflict and they will be applied to different versions. Therefore, only causally related operations can overwrite each other. Furthermore, operations are executed according to their causal order. Hence, any operation, $O_1$, can only overwrite
the effect of another operation $O_2$, if $O_1$ is generated after the execution of, $O_2$, i.e. $O_2 \rightarrow O_1$. The overwritten condition for GRACE is formally defined in Definition 3.3.

**Definition 3.3.** Overwritten ‘$>$’

Given any two operations that are applied to the same object, the effect of operation $O_a$ has been overwritten by the effect of operation $O_b$, denoted by $O_a > O_b$, iff:

1. $O_a \rightarrow O_b$, and
2. $Att.Type(O_a) == Att.Type(O_b)$. □

The effects between operations which do not overwrite each other, such as operations targeting different objects or which change different attributes, are commutative. Their execution does not have any impact on the effect of other commutative operations. Hence, undoing an operation also has no impact on its commutative operations. So commutative operations do not need to be considered during undo. Therefore, the rest of this section will only refer to operations which are applied to the same object and change the same attribute.

The effect of an operation might be visible or not visible depending on whether the effect of this operation has been overwritten or not. An operation’s effect is visible on the object it was applied to if the attribute of this object has the value specified by the operation. Immediately after an operation is executed, its effect is visible. This operation’s effect becomes not visible when its effect has been overwritten (or replaced) by the effect of another operation. The definition of the visible effect of an
operation is as in Definition 3.4.

**Definition 3.4.** Visible effect

The effect of an operation is *visible* on an object if the attribute of this object has the value specified by the operation.

The effect of executed operations, visible or not, exist in the document state in the sense that they will participate in DO, undo, or redo of other operations. After operations have been undone, it is assumed they will not be re-executed or participate in DO, undo, or redo of other operations, so their effects are permanently eliminated, unless they themselves are redone. Hence, to undo an operation, we only need to ensure its visible effect is eliminated at the time of undo. After operations have been redone, it is assumed their effects exist in the document state as if they have never been undone, even thought their effects may or may not be visible at the time of redo. Hence, to redo an operation, we only need to ensure the correct visible effect is produced at the time of redo.

In order to understand the impact of visible and overwritten effects on undo/redo, consider the following scenario:

**Example 3.1.**

Given any three operations, $O_1$, $O_2$, and $O_3$, changing the same attribute, say *color*, with values of *red*, *green* and *blue* respectively. Assuming they have the overwritten relationship of $O_1 \succ O_2 \succ O_3$, then they must have the causal ordering of $O_1 \rightarrow O_2 \rightarrow O_3$ and be executed in that order. At the state $S(O_1 \circ O_2 \circ O_3)$, only the
effect of $O_3$, blue, is visible.

Any of these three operations in Example 3.1 can be selected for undo and any undone operation can be selected for redo. What should happen when operations whose effects have been overwritten are selected for undo? At the state $S(O_1 \circ O_2 \circ O_3)$, the effects of $O_1$ and $O_2$ have been overwritten. There are three cases:

- $O_1$ is selected for undo. At the state $S(O_2 \circ O_3)$, the color remains blue.
- $O_2$ is selected for undo. At the state $S(O_1 \circ O_3)$, the color also remains blue.
- Both $O_1$ and $O_2$ are selected for undo. At the state $S(O_3)$, the color still remains blue.

So undoing $O_1$ and $O_2$ should not produce any visible effect, i.e. the attribute values of this object are the same before and after undo.

Generally, this effect can be specified as in GU1. Undoing an operation, $O$, that has been overwritten will have no visible effect. This satisfies Definition 3.1 because the effect of $O$ has already been eliminated. Furthermore, making no change ensures that the effects of the rest of other operations will retained in their execution order.

**Graphics Undo rule 1 (GU1):** Given operation $O$ where the effect of $O$ has been overwritten on an object $G$, undoing $O$ will produce no visible effect on $G$.

So, what if the operation to be undone has not been overwritten? At the state $S(O_1 \circ O_2 \circ O_3)$, $O_3$ has not been overwritten. Undoing $O_3$ would result in the state
$S(O_1 \circ O_2)$ with the effect of $O_2$, being visible. This special relationship between $O_3$ and $O_2$ is not simply that $O_3$ overwrites $O_2$, because $O_3$ also overwrites $O_1$. It would be incorrect to produce the effect of $O_1$ when undoing $O_3$. The precise relationship between $O_2$ and $O_3$ is that $O_2$ is the operation that is most recently overwritten by $O_3$, as defined in Definition 3.5.

**Definition 3.5.** Most recently overwritten `$\succeq$`

Given two operations, $O_a$ and $O_b$, applied to an object represented by the compatible group $CG$, $O_b$ is the operation most recently overwritten by $O_a$, denoted by $O_b \succeq O_a$, iff:

1. $O_b \in CG \land O_b \succ O$, and
2. $\neg \exists O_c \in CG \land O_b \succ O_c \land O_c \succ O_a$.

**Graphics Undo rule 2 (GU2):** Given any operation $O$ with visible effect on an object $G$, undoing $O$ will produce the effect of $O_x$ on $G$, where $O_x \succeq O$.

GU2 satisfies Definition 3.1 because by producing the effect of $O_x$, the effect of $O$ will be eliminated. Since $O_x$ is now the last operation applied to that attribute, $O_x$ overwrites the effect of other operations applied to that attribute. The effects of these operations are still retained even though they are not visible. This is because any of these operations’ effect will become visible when all operations which overwrite it have been undone.

The effects specified by GU1 and GU2 also satisfy the undo property in Definition 3.2. This can be illustrated by undoing $O_2$ and $O_3$ from Example 3.1, in any
order, as shown in Example 3.1.1 and 3.1.2.

**Example 3.1.1:**
Undoing $O_3$ then $O_2$.

1. Undoing $O_3$ will produce the effect of $O_2$, since $O_2 \succeq O_3$, resulting in $S(O_1 \circ O_2)$.
2. Undoing $O_2$ will produce the effect of $O_1$, since $O_1 \succeq O_2$.

So the effect of $S(O_1)$ is produced at the end.

**Example 3.1.2:**
Undoing $O_2$ then $O_3$.

1. Undoing $O_2$ will produce no visible effect, since $O_2 \succ O_3$, resulting in $S(O_1 \circ O_3)$.
2. Undoing $O_3$ will produce the effect of $O_1$, since $O_1 \succeq O_3$.

The final effect is also $S(O_1)$.

In both examples, the same state of $S(O_1)$ is arrived at after undoing $O_2$ and $O_3$ in different order. Hence, the undo property is satisfied.

The redo effect should be consistent with the DO effect. Since DO respects the causal order, then redo should also respect the causal order. At the state $S(O_1 \circ O_3)$, if $O_2$ is to be redone, then the state $S(O_1 \circ O_2 \circ O_3)$ should be reached since $O_1 \rightarrow O_2 \rightarrow O_3$. At this state, $O_2$ has been overwritten, redoing $O_2$ does not change the exiting visible effect of the object. Even though the effect of $O_2$ is
not visible after redo, its effect is now in the document state because, when all operations which overwrite \(O_2\) have been undone, then the effect of \(O_2\) will become visible.

**Graphics Redo rule 1 (GR1):** Given any operation \(O\) targeting object \(G\), if the state of \(G\) contains an operation that overwrites \(O\), then redoing \(O\) will produce no visible effect on \(G\).

In the next situation, at the state \(S(O_1 \circ O_2)\), redoing \(O_3\) would reach the state \(S(O_1 \circ O_2 \circ O_3)\). In this state, only the effect of \(O_3\) is visible. Generally, this effect is specified as in **GR2**.

**Graphics Redo rule 2 (GR2):** Given any operation \(O\) targeting object \(G\), if there is no operation in the state of \(G\) that overwrites \(O\), then redoing \(O\) will produce the visible effect of \(O\) on \(G\).

Redo can be regarded as undoing an undone operation. Redoing an operation that has just been undone should reverse the undo effect. The GU and GR rules satisfy this condition. Undoing an operation that has been overwritten (GU1) produces no visible effect. Redoing this operation (GR1) also produces no visible effect. Undoing an operation with visible effect (GU2) will eliminate its visible effect. Redoing this operation (GR2) will restore its visible effect.

Redoing undone operations in any order should restore the document to the state before these operations were undone. This can be illustrated by redoing operations \(O_2\) and \(O_3\), from Example 3.1.1 and 3.1.2, in any order.
Example 3.1.3:
Redoing $O_2$ then $O_3$.

1. Redoing $O_2$ will restore the effect of $O_2$ which overwrites the effect of $O_1$.

2. Redoing $O_3$ will restore the effect of $O_3$ which overwrites $O_2$.

The end result, at the state of $S(O_1 \circ O_2 \circ O_3)$, is that only the effect of $O_3$ is visible.

Example 3.1.4:
Redoing $O_3$ then $O_2$.

1. Redoing $O_3$ will restore the effect of $O_3$ which overwrites $O_1$.

2. Redoing $O_2$ will have no visible effect, since $O_2 \succ O_3$.

The final result is that only the effect of $O_3$ is visible.

In both examples, the state in Example 3.1 is produced after redoing undone operations in different orders. This property is consistent with the undo property in Definition 3.2.

These four rules, $\textbf{GU1}$, $\textbf{GU2}$, $\textbf{GR1}$ and $\textbf{GR2}$, define the undo/redo effects for any graphics operation. The effect specified by these rules has the same effect as if all operations in the document state are executed in their causal order. Given any document state containing only dependent operations, the effect of this document is unique. Therefore, after undo/redo when the document state is the same at all sites, convergence is ensured.
3.4.2 Undo/redo effect generation

This section examines how to produce the undo/redo effects specified by the GU and GR rules. Given an object represented by a compatible group, $CG$, what effect should $CG$ have after undoing/redoing an operation $O$ in $CG$? The solution will be represented by a special operation $O'$ such that when $O'$ is applied to $CG$ at its current state, the correct undo/redo effect of $O$ will be produced (as defined by the GU and GR rules).

The $UNREG$ (UNdo/Redo Effect Generation) algorithm is defined in Algorithm 3.1 for producing the undo effect by generating $O'$. $UNREG$ takes an operation, $O$, to be undone/redo, and a compatible group, $CG$, representing the object that $O$ is (to be) applied to. $UNREG$ returns $O'$ which will produce the correct undo/redo effect when applied to $CG$ at its current state. In the case of where there is an operation in $CG$ which overwrites $O$, $O'$ is assigned an identity operation, $I$ (as shown below in Step 1). Applying $I$ to an object will not change any attribute value of this object.

Algorithm 3.1. $UNREG(O, CG) : O'$

1. If $(O_x \in CG \land O \succ O_x)$ then $O' := I$;
2. Else
   (a) If $O$ is being redone then
   $O' := O$;
   (b) Else $O' := MostRecentlyOverwrittenBy(O)$;

Step 2 of $UNREG$ deals with the situation where $O$ has not been overwritten. If $O$ is being redone, then it simply reproduces the effect of $O$. Hence, $O'$ is assigned
Otherwise, for undoing $O$, $O'$ should be assigned the values of an operation $O_x$, where $O_x \geq O$. This is only a general guideline and the details may vary depending on the semantics of the operations as discussed in Section 3.8. For now, we will simply assume the function $MostRecentlyOverwrittenBy(O)$ will return the operation such that, when this operation is executed in the current state, the required undo visible effect will be produced.

### 3.4.3 Undo operation representation

After executing user generated operations, they are stored by appending them to History Buffer (HB). The document state at a particular time can be derived by executing all operations in HB from the beginning of the list till the end. However, the operation to produce the undo/redo effect, $O'$, is a system generated operation. What to do with $O'$ after it has been executed?

There are two methods to treat $O'$ after it has been executed. The first method is to append $O'$ to HB. In Example 3.1, undoing $O_3$ in the state $S(O_1 \circ O_2 \circ O_3)$, would result in $O'_3$, which contains the effect of $O_2$, to be appended to HB. Executing all operations in HB would produce the state $S(O_1 \circ O_2 \circ O_3 \circ O'_3)$, which is equivalent to $S(O_1 \circ O_2)$. However, by appending $O'$ in HB, $O'$ would require a valid state vector because it may be used for concurrency control. Since $O'$ is a system generated operation, it would be difficult to produce a valid state vector value for $O'$. Therefore, we have chosen the following approach.

In this second method, after $O$ has been undone, not only is $O'$ not stored, but also $O$ is removed from HB. In graphics editing systems, $O$ can simply be removed after it has been undone, because removing $O$ will not affect any operation executed after $O$ (unlike the text environment). In Example 3.1, undoing $O_2$ would result in
the removal of $O_2$ from HB. Executing all operations in HB would produce the state $S(O_1 \circ O_3)$, which is the correct state after undoing $O_2$. After undo, $O$ still needs to be stored somewhere so that it can be used for redo. Furthermore, after redoing $O$, $O$ should be inserted in the same HB position (relative to other operations) where it was removed from. For example, after redoing $O_2$, $O_2$ should be inserted between $O_1$ and $O_3$ so that executing all operations in HB would produce $S(O_1 \circ O_2 \circ O_3)$.

Due to these reasons, an implicit removal scheme is introduced. After execution, $O'$ is discarded. However, instead of removing $O$, $O$ remains in HB but is marked as undone. Any operation marked as undone will be regarded as having been removed from HB and will not appear in any compatible group (unless otherwise stated). To facilitate the implicit removal scheme, the following notations are introduced. Each operation has a binary field, called mark, to indicate whether the operation has been undone, i.e. if an operation has been undone then mark = true, otherwise mark = false. The function isMarked($O$) returns the mark value for the operation $O$. The function Mark($O$) inverses the mark value for $O$, i.e. if isMarked($O$) is false (or true) then Mark($O$) will set mark to true (or false).

3.5 Undo effect on multiple versions

The previous section looked at how to produce the undo/redo effect for objects. Due to the multiple object version concurrency control scheme, an object may be represented by multiple versions. How to ensure all versions have the correct undo/redo effect? Versions are created due to the execution of conflicting operations. Therefore, when undoing operations, it is possible that versions may need to be removed when conflicting operations are undone. A removed version may be recreated later due
to redo of some operations. This section will first look at how to produce the correct visible undo/redo effect on versions. Then the multiple version effect regarding undo/redo is examined. Finally, how versions are removed and recreated is presented.

3.5.1 Visible undo/redo effect on multiple versions

Due to the MOVIC concurrency control algorithm used in GRACE, multiple versions of an object may be created. Furthermore, an operation may be applied to more than one version. If the operation selected for undo/redo has been applied (or should be applied) to more than one version, then the correct undo/redo effect should be produced on those versions.

To illustrate the problems associated with undo/redo and multiple versions, consider the following scenario:

Example 3.2.

There are five operations. Their generation and execution order is as shown in Figure 3.1. Operations $O_1$, $O_2$, and $O_3$ are generated targeting the same object. Furthermore, $O_2 \odot_B O_3$. The execution of these three operations resulted in versions $CG_1 = \{O_1, O_2\}$ and $CG_2 = \{O_1, O_3\}$. $O_1$ is in both compatible groups because it is compatible with both $O_2$ and $O_3$. Then operation $O_4$ targeting $CG_1$ is generated, so it is only applied to $CG_1$. Also, operation $O_5$ is generated which targets $CG_2$, so it is only applied to $CG_2$. The final result is $CG_1 = \{O_1, O_2, O_4\}$ and $CG_2 = \{O_1, O_3, O_5\}$. The CRT for these five operations is as shown in Figure 3.2. No relationship is specified between $O_4$ and $O_5$ since $Target(O_4) \not\subseteq Target(O_5)$ nor $Target(O_5) \not\subseteq Target(O_4)$, so they will never be used to compare for conflict.
Figure 3.1: The time-space diagram for Example 3.2

![Time-space diagram](image)

Figure 3.2: The CRT for Example 3.2

<table>
<thead>
<tr>
<th>OP</th>
<th>(O_2)</th>
<th>(O_3)</th>
<th>(O_4)</th>
<th>(O_5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O_1)</td>
<td>(\otimes)</td>
<td>(\otimes)</td>
<td>(\otimes)</td>
<td>(\otimes)</td>
</tr>
<tr>
<td>(O_2)</td>
<td>(\otimes_I)</td>
<td>(\otimes_I)</td>
<td>(\otimes)</td>
<td>(\otimes_I)</td>
</tr>
<tr>
<td>(O_3)</td>
<td>(\otimes_I)</td>
<td>(\otimes_I)</td>
<td>(\otimes)</td>
<td>(\otimes_I)</td>
</tr>
<tr>
<td>(O_4)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

In the above example, some operations are applied only to one version, while some are applied to more than one. The first problem is how to find the versions an operation is applied to. The TOVER scheme can be used to determine a set of possible compatible groups, \(NCGS\), an operation, \(O\), has been applied to. However, \(O\) may not have been applied to all compatible groups in \(NCGS\). In Example 3.2, the \(NCGS\) for \(O_2\) and \(O_3\) will be \(\{CG_1, CG_2\}\). However, each operation is only applied to one compatible group. The solution is to first determine the compatible groups in \(NCGS\) which \(O\) has been applied to. Then only apply \(O\) to these compatible groups.
The second problem is that the undo/redo effect may be different on different compatible groups. In Example 3.2, $O_1$ is applied to both $CG_1$ and $CG_2$. It is possible that $O_1$ was overwritten by $O_4$. Since $O_4$ is only applied to $CG_1$, so the effect of $O_1$ has not been overwritten on $CG_2$. The solution to this problem is to generate $O'$ separately for each version.

The $MOP$ (Multiple Object version aPplication) algorithm is presented in Algorithm 3.2. $MOP$ takes an operation, $O$, to be undone/redoed and a set of versions, $NCGS_i$, which $O$ may have been applied to. $MOP$ requires that for any compatible group, $CG$, in $NCGS_i$, if $O$ is to be undone/redoed from $CG$, then $O$ should be in $CG$. For each $CG$ containing $O$, $UNREG(O, CG)$ is called to produce $O'$. Then $MOP$ invokes the $makeVisible(O', CG)$ function to make the effect of the inverse operation visible on the version represented by $CG$.

**Algorithm 3.2. $MOP(O, NCGS_i)$**

For each $CG \in NCGS$ do

   If $O \in CG$ then

   \[ O' := UNREG(O, CG); \]

   \[ makeVisible(O', CG); \]

It should be noted that $MOP$ only ensures the correct undo/redo effect is produced on the given versions. It does not make changes to any compatible group in its input $NCGS_i$. The tasks of ensuring the correct number of compatible groups, and the correct operations in compatible groups are examined in the next two subsections.
3.5.2 Multiple version effect

The MOVIC algorithm takes a NCGS with \((i-1)\) operations then adds an operation to produce another NCGS with \(i\) operations. The version removal algorithm should do the opposite of MOVIC; given a NCGS with \(i\) operations, an operation should be removed to produce another NCGS with \((i - 1)\) operations. Any NCGS, produced by either DO or undo, should satisfy the conditions of a normalized compatible group set as in Definition 2.4, which defines the multiple object version effect.

The multiple object version effect was originally defined for DO, where operations are executed following their causal order. This means versions are first created by the execution of conflicting operations before other operations targeting those versions can be generated and applied. If operations are undone in the reverse causal order, operations targeting a specific version will all be undone before the conflicting operation is undone and the version removed. However, operations may be selected for undo in any order (including in the causal order). What should happen when a conflicting operation which caused the creation of a version is undone before operations targeting that version are undone?

Consider the effect of undoing operations in Example 3.2. Suppose \(O_2\) is selected for undo. Removing \(O_2\) from the compatible groups results in \(CG_1 = \{O_1, O_4\}\) and \(CG_2 = \{O_1, O_3, O_5\}\). The existence of both compatible groups is required because if either of these compatible groups is removed, the effects of some operations will be lost. If \(CG_1\) is removed, then the effect of \(O_4\) will be lost. If \(CG_2\) is removed, then the effects of \(O_3\) and \(O_5\) will be lost. Therefore, both compatible groups should be kept.
O₂ is partly responsible for the creation of CG₁. So does keeping CG₁, after undoing O₂, preserve the undo effect of O₂? If O₂ is a Create operation, then undoing O₂ would certainly result in the removal of CG₁. However, O₂ is not a Create operation because Create operations do not conflict with any operation (and O₂ is a conflicting operation). Therefore, there is no obvious requirement that undoing O₂ should result in the removal of CG₁. The execution of O₂ must have caused the change of an attribute which resulted in the conflict. Then undo O₂ should change the attribute back to the previous value (if O₂ has not been overwritten). This result preserves the undo effect of O₂.

Does this effect satisfy the conditions for a normalized compatible group set? According to Definition 2.4, these compatible groups should exist only if there is an operation in CG₁ which conflicts with another operation in CG₂. Such operations exist, since O₄ indirectly conflicts with O₃. However, this indirect conflict relationship, is the result of O₂ directly conflicting with O₃. Even though O₂ has been undone, operations relying on O₂ to determine their indirect conflict relationships remain. Therefore, a undone operation may be required by other operations to determine their indirect conflict relationships.

Continuing with this example, suppose O₃ is selected for undo. Removing O₃ from the compatible groups results in CG₁ = \{O₁, O₄\} and CG₂ = \{O₁, O₅\}. Again, removing either CG₁ or CG₂ does not preserve the effect of all executed operations. Therefore, both versions should be kept. However, according to Definition 2.4, there is no operation in CG₁ which conflicts with an operation in CG₂, therefore, \{CG₁, CG₂\} is not a valid NCGS and they should be combined into one compatible
group \{O_1, O_4, O_5\}. This combined effect preserves the effects of all operations, however, applying these operations to the same object is not desirable for the following reasons:

- These operations are generated targeting different objects so they should not be applied to the same object.

- These operations may be independent and changing the same attribute to different values. They would be regarded as conflict if they target the same object.

- The users may have seen these operations’ effects on different objects. After undo, these effects are on the same object. This is obviously not what the user would expect from undo.

The cause of such a combined effect specification is due to the conflict definition assuming the operations compared for conflict have either the same scope (i.e. targeting the same object) or overlapping scopes. Operations with non-overlapping scopes are not included in the conflict definition because they are never compared for conflict during DO. However, operations with non-overlapping scopes may need to be compared for conflict during undo. In our example, \(O_4\) and \(O_5\) have different scopes of \(CG_1\) and \(CG_2\) respectively. Their scopes are non-overlapping. During DO, these operations will be applied to objects within their respective scopes so they will not be compared for conflict. However, when undoing an operation with both \(CG_1\) and \(CG_2\) in its scope (e.g. \(O_3\)), then operations in both \(CG_1\) and \(CG_2\) need to be compared for conflict (for possible version removal). Since \(O_4\) and \(O_5\) are regarded as compatible by the conflict definition, the effect specified by the normalized compatible group set is for \(O_4\) and \(O_5\) to be applied to the same object.
Since conflict is defined to determine when operations should be applied to different versions, operations targeting different objects should obviously be applied to different objects. Therefore, conflict definition should be extended to cover the situation of operations with non-overlapping scopes. In Section 2.5, indirect conflict was introduced to ensure consistent effects are produced when one operation’s scope is a subset of another operation’s scope. We will extend the indirect conflict definition to include the handling of operations with non-overlapping scopes. Operations with non-overlapping scopes target different objects. For two operations targeting different objects, $O_a$ and $O_b$, it must be that $\text{Target}(O_a) \not\subseteq \text{Target}(O_b)$ and $\text{Target}(O_b) \not\subseteq \text{Target}(O_a)$.

**Definition 3.6.** Extended indirect conflict relation

Given two independent operations, $O_a$ and $O_b$, $O_a$ and $O_b$ have the indirect conflicting relation, denoted by $O_a \otimes_I O_b$, if either:

1. there is an operation $O_c$ and compatible group $CG$ such that:
   
   - $O_c \otimes_D O_a$,
   - $\text{Target}(O_b) = \text{Id}(CG)$ and $O_c \in CG$, and
   - $\text{Target}(O_c) \subset \text{Target}(O_b)$, or

2. $\text{Target}(O_a) \not\subseteq \text{Target}(O_b)$ and $\text{Target}(O_b) \not\subseteq \text{Target}(O_a)$

In summary, this section has determined how indirect conflict checking, which involves an undone operation, should be carried out. Furthermore, an additional condition for indirect conflict was defined. As the result of these adjustments, the
definition for normalized compatible groups set specifies the multiple object version effect for both DO and undo. As proven in Section 2.2.4, the effect specified by the normalized compatible groups set is unique for any given group of operations. Therefore, if all sites produce the normalized compatible groups set effect for the same set of operations, then convergence is ensured.

3.5.3 Version removal

During DO, multiple versions may be created by MOVIC. Therefore, during undo, multiple versions may be required to be removed. To illustrate why version removal is required in order to satisfy Definition 2.4, consider Example 3.2 with only operations $O_1$, $O_2$ and $O_3$. After the execution of these operations, the following compatible groups are produced: $CG_1 = \{O_1, O_2\}$ and $CG_2 = \{O_1, O_3\}$. Suppose $O_3$ is selected for undo. Undoing $O_3$ will produce the compatible groups $CG_1 = \{O_1, O_2\}$ and $CG_2 = \{O_1\}$. At this state, there is no operation in $CG_1$ which conflicts with an operation in $CG_2$. Therefore, according to condition 2 of Definition 2.4, the remaining operations, $O_1$ and $O_2$, should be in one compatible group. This can be achieved by removing $CG_2$ so that only $CG_1 = \{O_1, O_2\}$ remains.

How to determine when versions should be removed in general? Let $NCGS_i$ be the normalized compatible group set for a set, $GO$, containing $i$ operations. Let $NCGS_{i-1}$ be the normalized compatible group set for $GO - \{O\}$. A compatible group set, $CGS_{i-1}$, can be derived by removing $\{O\}$ from every compatible group in $NCGS_i$. It can be proven that $CGS_{i-1}$ satisfies condition 1 (in Definition 2.4) for $NCGS_{i-1}$. 


Theorem 3.

Given an $NCGS_i$ for a set of operations, $GO$, $CGS_{i-1}$ (derived by removing \{O\} from every compatible group in $NCGS_i$), satisfies condition 1 of the normalized compatible groups set for the set of operations $GO – \{O\}$.

Proof: $NCGS_i$ satisfies condition 1. For any group of mutually compatible operations in $GO$, there must be at least one $CG \in CGS$, such that all these compatible operations coexist in $CG$. Removing \{O\} from $GO$ will not increase the group of mutually compatible operations. $CGS_{i-1}$ retains all compatible groups in $NCGS_i$, hence $CGS_{i-1}$ satisfies condition 1 of the normalized compatible groups set. \qed

However, $CGS_{i-1}$ may not satisfy condition 2 because $O$ may be a conflicting operation. By removing $O$, two compatible groups in $CGS_{i-1}$ may not have a pair of conflicting operations between them. It can be proven that if $CGS_{i-1}$ does not satisfy condition 2, then there must be a $CG_m$ and a $CG_n$ in $CGS_{i-1}$ such that $CG_m \subset CG_n$.

Theorem 4.

For any $CG_m \in CGS_{i-1}$, if $CG_m \oplus CG_n$ and $CG_n \in CGS_{i-1}$, then it must be that $CG_m \subset CG_n$.

Proof: $NCGS_i$ satisfies condition 2. For any $CG_m, CG_p \in NCGS_i$, there must be at least one $O_x \in CG_m$, and one $O_y \in CG_p$, such that $O_x \oplus O_y$. For the corresponding $CG_m, CG_p \in CGS_{i-1}$, to be mutually compatible, $O_x$ must be removed and $O_y \oplus (CG_m - \{O_x\})$. Then, according to condition 1, there must be a $CG_n \in CGS_{i-1}$ such that $CG_n = CG_m - \{O_x\} + \{O_y\}$. ($CG_n$ and $CG_p$ may or may not be the same.) Since $(CG_m - \{O_x\}) \subset (CG_m - \{O_x\} + \{O_y\})$, thus the theorem follows. \qed
Since subsets in a CGS are what causes the violation of condition 2, therefore, by removing all compatible groups which are subsets of another in CGS, condition 2 will be satisfied (according to equivalent CGS rule 2). Hence, a CGS can be transformed into a NCGS by removing all subsets within the CGS. Therefore, to convert CGS$_{i-1}$ into NCGS$_{i-1}$ simply remove any subsets from CGS$_{i-1}$. The MOVEM (Multiple Object Version Merging) algorithm defined in Algorithm 3.3 specifies how to obtain NCGS$_{i-1}$ given operation O and NCGS$_i$.

Algorithm 3.3. MOVEM(O, NCGS$_i$) : NCGS$_{i-1}$

1. NCGS$_{i-1}$ := \{ \}; \ M := \{ \};
2. Remove O from all CG \in NCGS$_i$;
3. For each CG \in NCGS$_i$ do
   If \exists C_{G_x} : CG \subseteq C_{G_x} then
      \ M := M + CG;
4. NCGS$_{i-1}$ := NCGS$_i$ - M;

Example 3.2.1:
From Example 3.2, operation O$_2$ is selected for undo. Executing O$_2$ via MOVEM results in CG$_1$ = \{O$_1$, O$_4$\} and CG$_2$ = \{O$_1$, O$_3$, O$_5$\}. No merging is performed because CG$_1$ \not\subseteq CG$_2$ and CG$_2$ \not\subseteq CG$_1$.

Example 3.2.2:
Following the execution in Example 3.2.1, O$_4$ is selected for undo. Removing O$_4$ results in CG$_1$ = \{O$_1$\} and CG$_2$ = \{O$_1$, O$_3$, O$_5$\}. Since CG$_1$ \subseteq CG$_2$, CG$_1$ is removed,
only $CG_2$ remains.

It can be shown that the results produced in Examples 3.2.1 and 3.2.2 are NCGSs. Once a version is removed from an NCGS, it is permanently discarded. This is because it is assumed that this version can always be re-created by MOVIC when needed.

3.5.4 Version re-creation

During undo, versions may be removed. Therefore, during redo, versions may need to be re-created. When to re-create removed versions? From the analysis in the previous section, a $CG_m$ will not be removed if there are two types of operations in $CG_m$:

- $O_x$ such that $O_x \otimes O_x$ where $O_x \in CG_n$ and $CG_m \neq CG_n$, or
- $O_y$ targeting $CG_m$ then $O_y$ can only be applied to $CG_m$.

Having either of these two types of operation in $CG_m$ will guarantee that $CG_m$ will not be a subset of any compatible group. Assuming $CG_m$ has been removed, if either of these two types of operation is to be executed/re-executed, then $CG_m$ needs to be re-created.

Since $O_x$ created the version $O_y$ is targeting, so it must be that $O_x \rightarrow O_y$. During DO, $O_x$ must be executed before $O_y$. MOVIC creates $CG_m$ when executing $O_x$, then $O_y$ is simply applied to $CG_m$. So $O_y$ is never used and cannot be used by MOVIC to create versions.

However, it is possible that, after the $CG_m$ has been removed, only $O_y$ is selected for redo. In this case, $CG_m$ without the effect of $O_x$ and $O_y$, i.e. $CG_m - \{O_x, O_y\}$, needs to be re-created for $O_y$ so the execution of $O_y$ will produce $CG_m - \{O_x\}$. Is
this meaningful, since $O_x$ caused the creation of $CG_m$? Yes, in undo, $O_x$ can be undone without undoing $O_y$. This is as shown in Example 3.2.1. This would leave $CG_m - \{O_x\}$ with the effect of all the operations compatible with $O_x$, including $O_y$. This should be the same effect produced by redoing $O_y$ without redoing $O_x$ first.

So how to produce $CG_m - \{O_x\}$ without re-executing $O_x$? One method is to execute $O_x$ via MOVIC, to produce $CG_m - \{O_y\}$, then remove the effect of $O_x$ to produce $CG_m - \{O_x, O_y\}$. Then $O_y$ can simply be applied to produce $CG_m - \{O_x\}$. However, to implement this approach, there are some issues that need to be resolved.

Firstly, given any operation, $O$, to redo, how to determine whether the version targeted by $O$ has been removed? If the version targeted by $O$ has been removed, then $TOVER(O)$ will not be able to find it, hence $TOVER(O) = \{\}$. Secondly, how to determine the operations that caused the creation of $O$'s target object (there may be more than one)? According to the object identification scheme, $COID$, any conflicting operations will have their operation identifiers added to the version's identification set for all the versions they are applied to. Therefore, these operations' signatures are written in $Target(O)$.

The operations that need to be applied to obtain $O$'s target version are the undone operations with their identifiers in $Target(O)$. Let $F$ denote the set of these operations. How to apply these operations? To start with, an operation $O_a$ in $F$ whose target version has not been removed needs to be found. This is so that a non-empty $NCGS$ from $TOVER(O_a)$ can be used by MOVIC to produce the next version targeted by another operation in $F$. Any operation $O_a$ in $F$ will have its target version available if there is no other operation in $F$ that is causally before $O_a$. This is because if $O_a$'s target version has been removed, then there must be an operation causally
before $O_n$ that caused the creation of $O_n$’s target version. If this operation has been undone, then it must be in $F$. The NCGS produced by $MOVIC(O_n, TOVER(O_n))$ is used to execute the next operation in $F$ which does not have an operation causally before it.

After executing all operations in $F$, there may be more than one version produced by $MOVIC$. The only version that is required for the execution of $O$ is the one with the identifier which matches $Target(O)$. Let $CG$ represent this version. The version that $O$ should apply to is $CG - F$.

**Algorithm 3.4. $VEP(O) : NCGS_{i-1}$**

1. $NCGS_{i-1} := \{ \}$.
2. Search $HB$ for a set of operations $F$ where 
   $$\forall O_x \in F : Id(O_x) \in Target(O) \land isMarked(O_x) = true.$$ 
3. Remove $O_y$ from $F : \neg \exists O_j \in F \land O_j \rightarrow O_y$.
   $$TMP := MOVIC(O_y, TOVER(O_y)).$$
4. Do until $F = \{ \}$ 
   Remove $O_y$ from $F : \neg \exists O_j \in F \land O_j \rightarrow O_y$.
   $$TMP := MOVIC(O_y, TMP).$$
5. Find $CG_{new} \in TMP$ corresponding to object $G$ :
   $$Id(G) = Target(O), \quad NCGS_{i-1} := \{ CG_{new} - F \}.$$ 

$VEP$ (VErsion Pre-creation) as described in Algorithm 3.4 is used to re-create removed versions. $VEP$ should be called by $MOVIC$ when an operation’s target object cannot be found by $TOVER$. This can only happen when its target object
has been removed. \( VEP \) takes this operation as the input and produces an \( NCGS \) containing only one compatible group. The compatible group is the version this operation should be applied to. The revised \( MOVIC \), which includes the use of \( VEP \) in Step 1, is as presented in Algorithm 3.5. This revised \( MOVIC \) will be able to re-execute operations in any order.

**Algorithm 3.5.** \( MOVIC(O, NCGS_{i-1}) : NCGS_i \)

1. If \( NCGS_{i-1} = \{ \} \) then \( NCGS_{i-1} := VEP(O); \)
2. \( NCGS_i := \{ \}; C := |NCGS_{i-1}|; \)
3. Repeat until \( NCGS_{i-1} = \{ \}; \)
   (a) Remove one \( CG \) from \( NCGS_{i-1}; \)
   (b) If \( O_i \odot CG \), then \( CG := CG + \{O_i\}; \)
   (c) Else if \( O_i \odot CG \), then \( C := C - 1; \)
   (d) Else
      • \( CG_{new} := O|(O \in CG) \land (O \odot O_i); \)
      • \( CG_{new} := CG_{new} + O_i; \)
      • \( NCGS_i := NCGS_i + CG_{new}; \)
   (e) \( NCGS_i := NCGS_i + CG; \)
4. If \( C = 0 \) then
   (a) \( CG_{new} := O_i; \)
   (b) \( NCGS_i := NCGS_i + \{CG_{new}\}; \)
5. For any \( CG_{new} \in NCGS_i \), if there is another \( CG \in NCGS_i \), such that \( CG_{new} \leq CG \), then \( NCGS_i := NCGS_i - \{CG_{new}\}. \)
Example 3.2.3:
Following the execution of Example 3.2.2 \((O_2 \text{ and } O_4 \text{ have been undone and } CG_1 \text{ removed})\), \(O_4\) is selected for redo:

1. The input \(NCGS\), produced by \(TOVER(O_4)\), is \(\{\\}\). Hence, \(VEP(O_4)\) is called.

2. \(VEP\) gathers \(F\), where \(F = \{O_2\}\) because \(Id(O_2) \in Target(O_4)\) and \(isMarked(O_2) = true\).

3. The initial \(NCGS\) for \(O_2\) is found by \(TOVER(O_2)\). \(CG_2\) is returned since \(Target(O_2)\) is a subset of the identifier of the object represented by \(CG_2\).

4. \(MOVIC(O_2, \{CG_2\})\) is called. \(CG_2 = \{O_1, O_3, O_5\}\), since \(O_2 \odot O_1, O_2 \odot D O_3\) and, \(O_2 \odot I O_5\), \(MOVIC\) adds a new compatible group called \(CG_1 = \{O_1, O_2\}\).

5. From the resulting \(NCGS\), \(\{CG_1, CG_2\}\), \(CG_1\) is selected because the object represent by \(CG_1\) has the identifier of \(Target(O_4)\). \(O_2\) is removed from \(CG_1\). Then \(CG_1\) is return to \(MOVIC\).

6. \(MOVIC\) applies \(O_4\) to \(CG_1\) which results in \(CG_1 = \{O_1, O_4\}\).

This result is correct because the same compatible groups are produced as in Example 3.2.1.

Example 3.2.4:
Following the execution of Example 3.2.3, \(O_2\) is selected for redo:

1. \(TOVER(O_2)\) produces a \(NCGS\), \(\{CG_1, CG_2\}\), which is to be used by \(MOVIC\).
2. Since $O_2 \circ CG_1$ so $CG_1 := CG_1 + \{O_2\}$. Hence $CG_1 = \{O_1, O_4, O_2\}$.

3. With $CG_2$, $O_2 \circ O_1$ but $O_2 \circ O_3$. Hence, a $CG_{new}$ containing $\{O_1, O_2\}$ is produced. However, $CG_{new} \subset CG_1$, therefore $CG_{new}$ is removed.

This result is correct because the same compatible groups are produced as in Example 3.2.

### 3.6 Object identification

The object identification scheme, $COID$, remains unchanged as in Definition 2.6. A conflicting operation's identifier is still used as a part of object identifier no matter whether this operation, or the operations it conflicts with, has been undone or not. This is because even though this operation may be undone, the versions it created may still exist. So versions may be removed or re-created, but their identifiers remain the same. By using the same identification scheme, $TOVER()$ can still be used to obtain the objects in the scope of an operation.

The execution and re-execution effect of an operation is limited to the objects within its scope. Therefore, $TOVER()$ is used for both DO and redo to obtain the initial $NCGS$. The visible effect of an operation can only appear on the objects within the scope of that operation. Therefore, the set of objects ($NCGS$) $MOP$ algorithm needs to work with can also be obtained by $TOVER()$.

To determine when a version should be removed during undo, the version an operation is applied to needs to be compared with other versions. However, in some situations, the scope of an operation may not have all the versions required for $MOVEM$ to perform version removal. In Example 3.2, when any operation is undone, the
versions that need to be compared for removal are \( CG_1 \) and \( CG_2 \). However, in Example 3.2.2, the scope of \( O_4 \) is only \( CG_1 \). Hence, MOVEM is not able to compare \( CG_1 \) and \( CG_2 \) to determine that \( CG_1 \) needs to be removed. Therefore, only providing the objects in an operation’s scope does not provide enough information for version removal.

This problem can be resolved by providing all versions made from the same object to MOVEM. The Object VErision Reconciliation (OVER) scheme in Definition 3.7 can obtain the set of versions made from the same object no matter which version the operation targets. Versions made from the same object must all have the Create operation’s identifier in their object identifier. Therefore, the identifiers of all versions made from the same object must intersect each other.

**Definition 3.7.** The OVER scheme

The NCGS produced by OVER for operation \( O \), denoted by \( TOVER(O) \), is a set of any \( CG \) corresponding to object \( G \) such that \( Target(O) \cap Id(G) \).

In Example 3.2, suppose \( O_1 \) is the Create operation. Therefore, \( Id(CG_1) = \{Id(O_1), Id(O_2)\} \) and \( Id(CG_2) = \{Id(O_1), Id(O_3)\} \). \( Target(O_2) = \{Id(O_1)\} \), hence, \( OVER(O_2) \) will return \( \{CG_1, CG_2\} \). \( Target(O_4) = \{Id(O_1), Id(O_2)\} \), hence, \( OVER(O_4) \) will also return \( \{CG_1, CG_2\} \). By using both compatible groups, MOVEM can now produce the correct undo effect for \( O_2 \) and \( O_4 \).

It should be noted that, in some situations, the OVER scheme may provide more versions than necessary. However, this will not cause any incorrect result since unrequired versions will simply be ignored by MOVEM and MOP. Furthermore, the number of versions for an object is expected to be small. Hence, this will not
have any significant impact on the performance of the system.

3.7 Undo and redo algorithms

Due to the different tasks required to be performed during undo and redo, two different algorithms, *Undo* and *Redo* in Algorithm 3.6 and 3.7, have been devised. When an operation, \( O \), is selected for undo/redo, an undo command, containing the identifier of \( O \), will be generated, executed locally, and propagated to remote sites. Executing a remote undo command requires that the operation be undone/redone to be determined. This can be done by searching for the operation in HB which matches the identifier specified in the undo command. Once \( O \) is found, if \( O \) is unmarked, then *Undo*\((O)\) is called. Otherwise *Redo*\((O)\) is called. It is important that the execution of *Undo* and *Redo* algorithms are done atomically in the sense that no any other operation is executed in between.

**Algorithm 3.6.** *Undo*(\(O\))

1. \(NCGS_i := \text{OVER}(O);\)
2. \(MOP(O, NCGS_i);\)
3. \(NCGS_{i-1} := \text{MOVEM}(O, NCGS_i);\)
4. \(Mark(O);\)
Algorithm 3.7. Redo($O$)

1. $NCGS_{i-1} := TOVER(O);$ 

2. $NCGS_i := MOVIC(O, NCGS_{i-1});$ 

3. $MOP(O, NCGS_i);$ 

4. $Mark(O);$ 

For Undo, $OVER$ is used to provide the initial $NCGS$ and $MOVEM$ is used for version removal. For Redo, $TOVER$ is used to provide the initial $NCGS$ and $MOVIC$ is used to re-create versions and determine which version an operation should be applied to.

$MOP$ is used by both algorithms. However, it is called in the inverse order of each other in relation to $MOVEM$ and $MOVIC$. This is due to the precondition required by $MOP$ where the operation that needs to be undone/redone should exist in the input $NCGS$. In Undo, $MOP$ is called before $MOVEM$ because after executing $MOVEM$, $O$ is removed from the $NCGS$. In Redo, $MOP$ is called after $MOVIC$ because $O$ is added to $NCGS$ after $MOVIC$ has been executed. Finally, the $Mark$ function is called at the end to mark or unmark the operation.

3.8 Operation specific issues in undo/redo

Section 3.4.2 described a general guideline on producing the undo effect. It is applicable to update operations. For Create and Destroy operations, and a combination of these and update operations, special treatments are required to ensure the correct inverse operations can be produced.
Normally, when an operation \( O \) is selected for undo, its inverse can be determined by finding \( O_x \) such that \( O_x \geq O \). However, if \( O \) is a \textit{Create} operation, then there is no operation whose effect was overwritten by \( O \). The inverse of a \textit{Create} operation is simply a \textit{Destroy} operation targeting the same object.

The next exception occurs when the update operation selected for undo overwrites the value set by a \textit{Create} operation, i.e. \( O_x \) is a \textit{Create} operation. For example \( O_x \) created \( G \) with color \textit{red} (and other attribute values). \( O \) is a \textit{Fill} operation and is applied to changed the color to \textit{blue}, \( O : \text{Fill}(G, \text{blue}) \). When undoing \( O \), it would obviously be incorrect to assign \( O' \) to be \( O_x \). What should be done here is to assign \( O' \) the same type as \( O \), but the value is derived from \( O_x \). So in this case \( O' : \text{Fill}(G, \text{red}) \).

Finally, the inverse operation that will be generated when undoing a \textit{Destroy} or redoing a \textit{Create} is the \textit{Create} operation. Executing this \textit{Create} will create an object and initialize all its attribute values. This may not be correct because if this object was updated before it was destroyed, then it should have the updated values when it is recreated (not its initial values). One solution is to re-execute all operations that were applied to that object. However, this solution requires that all operations which were applied to that object are to be kept. An easier solution is to keep the destroyed objects (and all their attribute values). This can be achieved by introducing a new attribute type called \textit{existence} to indicate whether an object has been destroyed or not. When an object is created, its \textit{existence} attribute is set to \textit{true} and this object is visible to the users. When an object has been destroyed, its \textit{existence} attribute is set to \textit{false} and this object is \textit{not} visible to the users (but is still kept). Even though a destroyed object cannot be seen, operations can still be applied to it. This will allow
all operations, especially the concurrent operations which arrived after the object has been destroyed, to be applied to their target objects. Hence, the destroyed object will have the effects of all operations targeting it. Therefore, when undoing a *Destroy* or redoing a *Create* operation, what should be done is to simply set the *existence* attribute of the destroyed object back to true. Since there is no operation which can only do this, a *Revive* operation is introduced. Applying a *Revive* operation to an object will only set its *existence* attribute to true. Therefore, during undo/redo, instead of generating *Create*, *Revive* is generated.

### 3.9 Comparisons

In the area of graphics editing systems, the only work that is close, in term of undo flexibility, to Any Undo is *direct selective undo* proposed by Berlage [4] (this is different to selective undo by Prakash and Shim). Direct selective undo allows any operation to be undone at any time with some exceptions, such as when the target object has been destroyed. In direct selective undo, when undoing operation $O$, $O'$ is assigned to the operation that was overwritten by $O$ when $O$ was executed. To undo $O$, $O'$ is applied to the current state, even if $O$'s effect has already been overwritten. For example, there are three *FILL* operations executed in the order of $O_1$, $O_2$ then $O_3$ which changed the color of an object $G$ to *red*, then *green* followed by *blue*. With direct selective undo, undoing $O_2$ would simply set $G$ to *red*, even though the effect of $O_2$ has already been overwritten. This effect violates the undo property and is incompatible with other undo models such as single-step undo, chronological undo, selective undo and Any Undo. Hence, direct selective undo cannot be used to implement other undo models.
Direct selective undo is not designed to work with any concurrency control protocol. It is not suitable for collaborative editing systems because inconsistency may arise due to the execution of concurrent undo operations in different orders. In the above example, \( O_2 \) and \( O_3 \) may be selected for undo concurrently, depending on the order of undo execution, \( G \) may be either red (if \( O_2 \) is applied last) or green (if \( O_3 \) is applied last). Therefore, the result is inconsistent when the operations are undone in a different order. In comparison, the only result that will be produced by Any Undo after undoing \( O_2 \) and \( O_3 \) in any order is red.

In the area of collaborative graphics editing systems, there are several authors who have listed undo/redo as their future work [9, 42, 53, 71]. However, as far as is known, there has not been a published result on user generated undo/redo for collaborative graphics editing systems. GRACE is the only graphics editing system to support Any Undo. Furthermore, GRACE is also the only system with the multiple object version concurrency control protocol which is able to remove and re-create versions due to undo/redo. Tivoli [71] uses a concurrency control protocol similar to the multiple object version scheme. However, Tivoli does not support undo/redo.

The undo/redo effect produced by the UNREG algorithm can also be achieved by a series of system generated chronological undo/redo operations. Given \([O_1, O_2, O_3]\), undo \( O_1 \) can be realized by: (1) undo \( O_3, O_2 \), then \( O_1 \); (2) redo \( O_2 \) then \( O_3 \). A variation of this method is used by Karsenty and Beaudouin-Lafon [54] to implement serialization concurrency control protocol for a collaborative graphics editing system called GroupDesign. They combined this method with a masking relation to reduce the number of required undos/redisos. Their masking relation is similar to the overwritten relation. The overwritten relation can be regarded as an optimization of the
masking relation. Overwritten relations take into account the operation orders, which are able to totally eliminate the need for the chronological undos/ redos.

3.10 Conclusion

This chapter has presented a technique for undoing any operation at any time with guaranteed success for collaborative graphics editing systems. The main contributions are:

1. the definition of undo/redo effects for object graphics operations (GU and GR rules);

2. the UNREG algorithm for generating an operation which, when applied to an object at the current state, will produce the correct undo/redo effect on that object;

3. the MOP algorithm which ensures the correct undo/redo effect is produced for a given object/version;

4. the MOVEM algorithm which removes obsolete versions when operations are undone;

5. the VEP algorithm which re-creates merged versions so operations can be re-done.

These algorithms are combined to form the Undo and Redo algorithms.

The three algorithms, MOP, MOVEM and VEP, are designed especially for the multiple object version concurrency control protocol. The graphics undo/redo effects and the UNREG algorithms are designed so that they are independent of
the concurrency control protocol. They can be used with other concurrency control protocols or no concurrency control protocol (i.e. single-user editors). All algorithms described in this chapter have been implemented in GRACE. The implementation and user interface issues regarding Any Undo are discussed in Chapter 5.
Chapter 4

Maintaining consistency by conflict prevention

The multiple object version scheme presented in Chapter 2 deals with how to maintain consistency after conflicting operations have been generated. This chapter examines how to prevent the generation of conflicting operations. The solution presented in this chapter is built on the multiple object version scheme. Hence, the results in both chapters are designed to work together (collaboratively).

4.1 Locking

Locking is a technique which is widely used for concurrency control to maintain consistency in database systems [33]. The concept of locking can be easily understood by the users, i.e. if someone locks an object, other people cannot access it. Locking can also be used to maintain consistency in collaborative graphics editing systems. Once an object is locked by a user, no conflicting operation can be generated on that object. Hence, convergence and intention preservation are ensured.

Locking can be compulsory or optional. With compulsory locking, a lock must be obtained before operations can be generated to edit that object. For systems
which use locking as the main/only concurrency control mechanism, locking must be compulsory. Examples of collaborative graphics editing systems with compulsory locking include CoDiagram [10], Ensemble [76], GroupDraw [43], GroupGraphics [84], and GroupKit [42]. With optional locking [107], an editing operation can be generated on an object with or without first locking that object. Optional locking is suitable when users are working on different areas/objects most of the time and are not likely to intrude other users’ work as in collaborative graphics editing systems [11]. In comparison, optional locking has some advantages over compulsory locking. Optional locking saves the users from the burden of having to use locks before editing most of the time. Furthermore, the system’s overhead associated with locking is reduced.

There are two approaches to locking, pessimistic and optimistic. With pessimistic locking, when a user wants to edit an object, the system has to obtain an exclusive lock on that object before the user can edit that object. This approach has the disadvantage that the response time is slowed down by the time it takes to obtain the exclusive lock. To obtain an exclusive lock, synchronization between all sites is required. This synchronization time is nondeterministic, and can be large when the network is congested or the editing sites are far apart, which means slow system response time. To overcome this problem, some systems use optimistic locking. With this approach, the system assumes the exclusive lock will be granted. So editing operations generated are executed locally without delay. This means two or more users may concurrently edit the same object. To ensure consistency, only the operations from one user are kept, concurrent operations from other users are aborted. To abort an executed operation, a roll-back method is used to undo the effects of executed operations to be aborted. This solution works well for database systems. However,
having an operation undone by the system is undesirable because it violates intention preservation. Furthermore, there is the problem of which operation to abort. The system decides which operations are to be aborted, not based on what users want, but on unrelated information such as the vector timestamp of the operations. This does not assist users in resolving the conflict.

This chapter examines optional and responsive locking in GRACE [21]. The multiple object version scheme is the main concurrency control scheme in GRACE. This allows locking to be the secondary concurrency control scheme and hence optional. These two schemes work together as follows: without locking, consistency will still be maintained by the multiple object version scheme; with the use of locking, the amount of possible conflicts can be reduced. The responsiveness of locking is provided by optimistic locking. However, the challenge here is to provide good responsiveness without aborting any generated operations to preserve the effects of all operations. In order to incorporate locking into GRACE, concurrency control issues regarding locking operations need to be addressed. For example, how to solve inconsistency problems caused by concurrent locking operations targeting the same object?

This chapter is organized as follows: Section 4.2 examines the types of locks suitable for GRACE, locking operation generation, and inconsistency problems associated with locking; Section 4.3 addresses the issue of maintaining locking consistency; Section 4.4 discusses when locks stabilize; and finally, the conclusion is presented in Section 4.5.
4.2 Locking in GRACE

In collaborative graphics editing systems, graphical objects are the obvious and suitable choice for applying locking since editing operations are generated to edit objects. Locking objects can prevent conflicts from occurring on those objects. Therefore, object locks have been chosen as one of the locking operations in GRACE. An object locking operation contains one or more object identifiers which specify the locking targets.

The other type of lock is region lock. A region lock can be used to lock an area of the shared document. Region locks are useful because users can specify private working areas into which no other user can intrude. Once a region is locked, only the lock owner can modify, create or delete objects within that region. Conceptually, a region can be regarded as an object which contains a rectangular area (the region) and a list of objects within the region.

The term lockable item or simply item will be used in the following sections to represent either objects or regions which can be locked.

4.2.1 To lock or not to lock?

Locking before applying operations is compulsory in other collaborative graphics editing systems. However, locking is optional in GRACE. Locks can be generated implicitly or explicitly. Locks are generated implicitly if they are placed automatically by the system. This approach is commonly used when locking is compulsory. However, applying optional locks implicitly requires intelligent agents which can decide whether locks are required for a certain situation (i.e. another user is about to edit the same object the current user is editing). The domain of implicit lock generation
is outside the scope of this thesis. With explicit lock generation, the user has to explicitly place a lock on an item, just like issuing an editing operation. Therefore explicit lock generation is the focus of discussion in this thesis.

What benefit does the user derive by locking before editing? If a user has locked an item, then the system guarantees the user has the editing rights to that item. If a user has obtained an exclusive lock on an item, then no other user can edit that item, hence no conflict will occur on that item (until that item becomes unlocked). If a user does not lock an item before editing, then it is possible that another user may edit or lock that item. In addition to editing rights, locks can also act as a group awareness mechanism. Locking can be used to inform other users which part of the document a user is currently editing.

The terminology of lock ownership will now be introduced. If a user has a lock on an item, then that user owns the lock to that item, or simply that user owns that item. If a user owns an item, than that user has the editing right to that item.

4.2.2 Responsive operation generation

With locking in place, before an operation can be generated, the request to generate that operation needs to be validated. This is to prevent the generation of an operation to edit an item locked by other user(s). A user’s request is valid if its target item is either unlocked or s/he owns the lock on this item. Once a request has been validated, an operation is generated. Invalid requests are rejected, and the user is informed.

How to determine if an item is locked or not? Conceptually, each item has an attribute which indicates if that item is locked and by whom. Each site maintains a list of all items. By finding an item in this list, the locking status of this item at a site can be determined.
Ideally, a user's request should be valid if at ALL sites the target item is either locked by this user or this item is unlocked. If a user owns the lock of an item at the local site, then this user will own the lock of this item at all sites. In this case, validation for any request by this user on this item can be done by checking the local locking status of the item. However, if an item is unlocked at the local site, it does not mean that this item is unlocked at all other sites because this item may be concurrently locked at remote sites. Under this condition, to validate a user request on an item which is unlocked at the local site, synchronization is required to determine if this item is also unlocked at other remote sites. This would slow down the validation process and thus the response time. In order to achieve fast response time, the synchronization in the validation process needs to be eliminated. Without synchronization, only the local locking status is known. Therefore, the validation condition is reduced to require only the item to be unlocked locally, as stated in Definition 4.1.

**Definition 4.1. A valid request**

Given an editing or locking request $Q$ generated by a user from site $j$, to edit/lock item $I$, $Q$ is valid if either this user owns the lock on $I$ or $I$ at site $j$ is not locked. □

With this definition of a valid request, the validation process checks only the local document to determine if a request is valid. This means global synchronization is not needed in the process of request generation, request validation, operation generation, and local operation execution. Only after these steps, is network communication used to broadcast the operation to all remote sites for execution. Hence, fast response time is ensured. It should be noted that these steps, from request generation till operation broadcasting, are done atomically at the local site. The process of operation
Figure 4.1: The operation generation process

generation is as shown in Figure 4.1.

4.2.3 Inconsistency problems

Let \( I \) be an item which is unlocked at all sites. Let \( O_1 \) be a locking operation generated from Site 1 to lock \( I \). Another operation, \( O_2 \), is generated independent of \( O_1 \) by a different user from Site 2 to edit or lock \( I \), as shown in Figure 4.2. In this situation, if no special action is taken, inconsistency would occur. If both are locking operations, then they would be first executed locally, then sent to remote sites for execution. However, when these operations arrive at the remote sites, \( I \) has already been locked by the other user. As the result, the operations would have to be aborted. This would result in divergence where, at Site 1, \( I \) is locked by \( O_1 \), and at Site 2, \( I \) is locked by \( O_2 \). A similar problem would occur if \( O_2 \) is an editing operation. \( O_2 \) would be applied at Site 2, but has to be aborted in Site 1 because \( I \) has already been locked.

Divergence can be resolved by using serialization. However, serialization would lead to intention violation. This is because to achieve serialization, the effect of one operation needs to be undone after it has already been executed at the local site. For example, if the serial effect is that \( I \) should be locked by \( O_1 \) instead of \( O_2 \), at Site 2, \( I \) is already locked by \( O_2 \) when \( O_1 \) arrives, so \( O_2 \) will need to be undone to allow \( O_1 \).
Figure 4.2: *Independent operations* $O_1$ and $O_2$ *may cause inconsistency problems* to lock $I$. Therefore, the effect of $O_2$ is not preserved.

In the next section, an instant locking scheme which ensures locking consistency will be presented.

### 4.3 Instant locking scheme

How to apply and preserve the intention of independent operations targeting the same item, where there is at least one locking operation? The first requirement in preserving the intention of operations is that once an operation is generated it has to be applied at all sites. Then, once an operation is executed, its effect is preserved even after the execution of all independent operations.

In order to examine how to preserve the effects of operations and also maintain convergence, consider the application of this approach to the same example in Figure 4.2. $O_1$ is a locking operation and $O_2$ can be either an editing or locking operation, so there are two situations at Site 1 after $O_1$ has been applied to $I$:

1. $O_2$ is an editing operation. At Site 1, $I$ is first locked by $O_1$. When $O_2$ arrives at Site 1, it has to be applied to $I$ to preserve its effect even though $I$ is locked.
2. $O_2$ is also a locking operation. When $O_2$ arrives at Site 1, $I$ is already locked by $O_1$. We propose that the ownership of the lock be shared between the users at Site 1 and Site 2. With this approach, the effects of both operations are preserved in $I$.

Convergence is also ensured by such execution because in both scenarios, the execution of operations in the order, $O_2$ followed by $O_1$ (as in Site 2), will produce the same effect as executing $O_1$ followed by $O_2$ (as in Site 1). This is because a locking operation is treated as commutative with other locking or editing operations.

In the first locking effect, after a user has locked an item, that item may be modified by other users. In the second locking effect, after a user has locked an item, s/he may have to share the ownership of that lock with other users. One may question, are the above locking effects reasonable? In this example, $O_1$ and $O_2$ are independent operations, and ONLY because of this will such a situation occur. It is impossible to have an operation $O_3$ where $O_1 \rightarrow O_3$ and $O_3$ is generated by a different user from $O_1$ to edit/lock $I$. Such a request would be invalid. Therefore, after a user has locked an item, operations from other users can be applied to that item only if they are independent of the local locking operation. After all independent operations have been applied, only then can the lock owners edit/lock that item. This period, starting from when a locking operation is executed until all operations independent of that locking operation are executed, is called the unstable period. During this period, a lock is said to be unstable. After the unstable period, a lock becomes stabilized. If users are informed when the lock is unstable and when the lock has stabilized, the effects encountered during the unstable period would not be unexpected. Furthermore, the users should understand the nature of shared locking. Locking does not guarantee
exclusive ownership, but it restricts the access rights to an item with a high probability that the access right is restricted to only one user (when the lock stabilized), in which case the lock is exclusive.

**Definition 4.2. Unstable period**

Let \( I \) be any unlocked item and \( O \) be any locking operation generated at Site \( j \) to lock \( I \). The *unstable period* of the lock on \( I \) starts when \( O \) is applied to \( I \) at Site \( j \) until all operations independent of \( O \) have been executed at Site \( j \).

While a lock is unstable, the number of owners of this lock can increase due to the application of independent locking operations. It is also possible for editing conflicts to occur on an object while its lock is unstable (the locking effect for this situation is discussed in Section 4.3.2). After this lock has stabilized, the number of lock owners cannot increase. Only the lock owners can edit or unlock this locked item. The number of owners decreases when an unlock operation is applied to this item. If there is only one owner of this item, then the lock is exclusive and no conflict will occur on it.

With this locking scheme, when a locking operation is generated, the user who generated the operation will gain ownership to the target item instantly. Therefore, this locking scheme is called *instant locking scheme* and the lock placed by it is called *instant lock*. The next two subsections address some specific issues associated with this locking scheme.
4.3.1 Instant lock sharing

This section discusses the details of lock sharing. What should the lock ownership be for independent locking operations whose target item is the same or overlaps? For independent object locking operations targeting the same object, the users who generated those operations will share the ownership on that object. For independent region locks with overlapping regions, the ownership for overlapping regions will be shared and non-overlapping regions remain exclusive. For example, consider two target regions, $R_1$ and $R_2$, with an overlapping area of $P$. Only the ownership of $P$ will be shared and the ownership for the rest of $R_1$ and $R_2$ remains exclusive, as shown in Figure 4.3.

Lock ownership for these two situations is obvious. However, what should the lock ownership be if there are independent object and region locking operations where the object $G$ is inside the target region $R$ (as shown in Figure 4.4)? Let $U_R$ be the set of owners who generated the region locking operations and $U_G$ be the set of owners who generated object locking operations. The lock ownership of $G$ and $R$ is as follows:

- All users in $U_R$ and $U_G$ should own $G$ because $G$ is inside $R$ (or partially inside).
Figure 4.4: *Independent object and region locks where the target object G is inside the region R*

- Only the users in $U_R$ should own $R$ because users in $U_G$ did not request the region lock.

Since $G$ is within a region lock, its behaviour is different from other locked objects. The effect on $G$ is as follows:

- All users in $U_R$ can edit $G$ and can move $G$ within $R$.

- All users in $U_G$ can edit $G$ but cannot move $G$ within $R$, since $U_G$ does not have access to $R$.

- All users in $U_R$ and $U_G$ can move $G$ outside of $R$.

- After $G$ has been moved outside of $R$ (i.e. at the completion of drag and drop) then $U_R$ lose their lock ownership of $G$. This is because the ownership of $U_R$ on $G$ is solely due to $G$ being in $R$. So if $G$ is outside $R$, then it is not within the region lock.
4.3.2 Instant locking and concurrent versions

For any object $G$ with a shared lock or whose lock is in the unstable period, conflicts may occur on $G$. Conflict will result in concurrent versions. What should the lock ownership be for these versions of $G$?

The first approach is to let the users who owned the lock on $G$ own the locks on all versions of $G$. For example, let $U_1$ and $U_2$ be the users who share the lock on $G$. $U_1$ and $U_2$ generated conflicting operations $O_1$ and $O_2$ respectively to edit $G$. Versions of $G$ will be made, the version $G_1$ for the application of $O_1$ and the version $G_2$ for the application of $O_2$. Both $U_1$ and $U_2$ will own the lock on both $G_1$ and $G_2$. The end result is that the locks on $G_1$ and $G_2$ are still shared (as with $G$).

The second approach is to let the users who caused the creation of the versions own different versions according to which object their operation is applied to. For example, users $U_1$ and $U_2$ issued conflicting operations $O_1$ and $O_2$ on $G$. $O_1$ is applied to $G_1$ and $O_2$ is applied to $G_2$. Then $U_1$ will own the lock on $G_1$ and $U_2$ will own the lock on $G_2$. So the locks on both versions are exclusive.

Which approach is better? The goal of locking is to reduce conflict by (eventually) granting a user exclusive access to an item. With the first approach, the lock on the versions are shared and conflicts may still occur on these versions. With the second approach the lock on the versions are exclusive and no conflict will occur on these versions (until they are unlocked). Therefore, the second approach is more desirable.

The example for the second approach works because each lock owner generated a conflicting operation which is applied to a different version. As a result, the lock ownership can be determined by the versions their operations are applied to. However, it is possible that some lock owners may generate operations which are applied to
more than one versions, and some may not generate any operation at the time. What should happen to their lock ownership?

Let $U_3$ be the user who also owns the lock on $G$ (in addition to $U_1$ and $U_2$). $U_3$ generated an operation $O_3$ which is independent and compatible with $O_1$ and $O_2$. $O_3$ will be applied to both $G_1$ and $G_2$. So which version’s lock should $U_3$ own?

To reduce the number of shared locks, $U_3$ can simply own the lock to one of the versions. Which version $U_3$ owns does not matter as long as it is the same at all sites.

However, operations may arrive in different order at different sites. Without waiting for all independent operations to arrive, different objects may be selected at different sites. For example, let the subscript attached to the operation also denote the site identifier where the object is generated, i.e. $O_i$ is from Site $i$. Let $U_3$ own the lock of the version made for the operation whose site identifier is the smallest, i.e. $G_1$. With only two conflicting operations, the correct object can always be selected. However, what if there is another operation, $O_4$, which conflicts with $O_1$ and $O_2$? $O_2$ and $O_4$ arrive first at Site 3 (where $U_3$ is), then versions $G_2$ and $G_4$ will be created. Now $G_2$ would be incorrectly selected as the locked object for $U_3$ because $2 < 4$. When $O_1$ arrives, the system can change $U_3$’s ownership from $G_2$ to $G_1$. However, this is too late because in the mean time, $U_3$ could have generated operations to edit $G_2$. This is wrong because $U_3$ is not suppose to have access to $G_2$.

Selecting an object for $U_3$ based on any condition is going to have the same problem. The only solution is to let $U_3$ own all the versions which may be selected. With this approach, the versions which may be selected are the only ones which $O_3$ is applied to. So if $O_3$ conflicts with $O_4$ then $U_3$ will own the lock of $G_1$ and $G_2$ but not $G_4$. 
Some lock owners may not generate any operation on $G$ while these conflicting and compatible operations are being generated. What should happen to these users’ lock ownership on versions of $G$? As with the previous case, simply choosing a version would be incorrect. These users did not generate any operation, so selecting the version their operations are applied to is out of the question. The only option is to let these users own the lock of all versions of $G$.

In summary, let $S$ be a set of independent operations all targeting the locked object $G$. Assume there is at least a pair of conflicting operations in $S$. For any user $U_G$ who owns the lock on $G$, after executing all operations in $S$:

- If $U_G$ generated an operation $O \in S$, then for any version, $G'$, of $G$ which $O$ is applied to, $U_G$ will own the lock on $G'$.

- If $U_G$ did not generate any operation in $S$, then $U_G$ owns the lock for all versions of $G$.

### 4.4 When does a lock stabilize?

How to determine when a lock has become stabilized? How would a site know when all operations independent of the locking operation have been executed at that site? These problems are similar to the garbage collection problem described by Sun et al for REDUCE [105]. The solution is based on the assumption that the underlying network is reliable and order-preserving between any pair of sites (e.g. TCP). This means if a sequence of operations is sent from the same site to the same destination, then these operations will arrive at their destination in the sending order.

The basic approach is that whenever a site, $j$, executes a remote locking operation $O$, Site $j$ needs to send a message to all remote sites telling them that Site $j$ has
executed $O$. If any site, $k$, receives this message from Site $j$ then Site $k$ knows that any operation independent of $O$ from Site $j$ must have already arrived and been executed at Site $k$ (because of the order-preserving nature of TCP connection). If there are $N$ sites, and Site $k$ has received $N - 1$ messages (excluding itself), then operations independent of $O$ from all sites must have already arrived and been executed at Site $k$.

The actual implementation can be done by simply checking dependency of the operations by comparing their state vectors. After Site $j$ has executed $O$, it sends a message, $M$, containing the state vector of Site $j$ to all sites. By comparing the state vector of $O$ with $M$, it can be determined that $O \rightarrow M$. So all operations from Site $j$ independent of $O$ must have already arrived. Hence, for any item $I$ locked by operation $O$, the lock on $I$ at Site $j$ becomes stable iff Site $j$ has received an operation dependent on $O$ from all participating sites.

### 4.5 Conclusion

Inconsistency caused by conflicting operations is a typical problem for distributed groupware systems. This chapter has presented an optional locking scheme which can reduce the number of conflicts that may occur in collaborative graphics editing systems. This locking scheme is used to support the multiple object version consistency maintenance scheme. Unlike other systems, where locking is compulsory, this locking scheme is optional. Optional locking has the performance advantage that locks are placed only when needed, which allows for maximum concurrent editing. Furthermore, the overhead associated with locking is reduced. Another feature
of this locking scheme is its fast response time. This is achieved without requiring any generated operation to be aborted (and therefore satisfies intention preservation) unlike other optimistic locking schemes.

Two different types of locks are proposed: object and region. The purpose of locking objects is obvious. A region lock can be used to lock an editing region. A locked region can ensure not only the objects inside the region will not be changed by other users, but also that no object will be created or moved into that region by other users.

GRACE is the first graphics collaborative editing system to adopt optional locking. However, optional locking was first proposed for REDUCE [107] (which is a collaborative text editing system). These two systems take similar approaches to optional locking, but also solve different context-specific problems related to text and graphics domains. The problem both systems need to solve occurs when independent operations target the same item. Shared locking is used in both systems to solve this problem. Each system also has to address the issues associated with the underlying concurrency control protocol. In REDUCE, locking operations need to go through operational transformation, just like other editing operations, to determine the correct locking region. In GRACE, locking needs to work in conjunction with the multiple object version scheme. This combination allows shared locks to take advantage of the creation of multiple versions to reduce the number of lock owners.

The behaviour of locks between text and graphics editing systems are quite different due to different nature of the document. For example, locking a segment of text in text editing systems may resemble locking a region in graphics editing systems, however, there is one major difference. Characters (including space) in a text
document are selected in contiguous order (i.e. characters 1, 2, ..., \(n-1, n\)). Hence, a
lock can only lock characters in contiguous order. Whereas region locks in graphics
editing systems can lock objects which are next to each other, no ordering is imposed
on these objects. For example, in a text document character 'A' is directly above
character 'B' which means they are in different lines. To select and lock 'A' and 'B'
in the same lock, this lock would have to contain all characters between 'A' and 'B'.
If 'A' and 'B' are graphical objects in graphics editing systems, they can be selected
and locked in one region lock without including other objects.

A well-designed user interface is important for locking. Users have to be able to
distinguish whether an item is locked or not. If it is, it should clearly display the
locking status and, if locked, by which user(s). The locking implementation issues,
including user interface issues, will be discussed as a part of the next chapter.
Chapter 5

Implementation issues

All algorithms and schemes presented in the previous chapters have been implemented in the GRACE prototype system in the programming language Java. GRACE is developed to test the feasibility of the proposed schemes and algorithms. Furthermore, building GRACE allowed exploration of system design and implementation issues. This chapter will discuss important implementation issues encountered during the development of GRACE.

5.1 The site server process

At the core of GRACE is the Site Server (SS) process. An SS process is run at each participating site. SSs on different sites communicate amongst each other via the Internet TCP/IP protocol.

There are multiple concurrent threads inside the SS. There is one thread to handle local user inputs (UIH), which is eventually converted into local operations. In addition, there is one thread (RSH) per each remote site dedicated to receive operations generated from that site. The central component in SS is the RE object which glues all the threads together (RE stands for REDUCE Engine because this design
is adopted from REDUCE [16, 105, 118]). All functions for executing operations and maintaining consistency are called from within RE. RE is implemented as a Java monitor object to provide the multiple threads with a synchronized point of access to the shared resources. Some of more important shared data include:

- the graphics objects list (GOL) which contains all the graphical objects in the document,

- history buffer (HB) which contains a list of executed operations, and

- local state vectors which contain the state vector values for the local site.

Each thread repeatedly executes the following two steps: waiting for operations from the corresponding remote/local site; then invoking RE to handle the incoming operation. When a thread invokes RE, this thread may be blocked (outside of RE) until there is no active thread inside RE. This will ensure operations are executed one at a time. This relationship between RE, UIH and RSH threads is shown in Figure 5.1.

### 5.2 Operation generation and execution

Local operations are generated from the UIH thread. Some operations, such as Create, Move, and Resize, may take sometime to generate due the dragging of the mouse cursor. While an operation is being generated, it should not be interrupted by the execution of remote operations. To prevent this interruption, at the start of operation generation (i.e. when the mouse button is pressed), UIH invokes a function in RE to set a variable, called mutex to true, before going back to handle operation generation. While mutex is true, any RSH thread trying to invoke RE will be blocked. When operation generation is completed (i.e. when the mouse button is released), mutex is
set to false. All RSH threads which were blocked will be activated to continue their execution.

Once a local operation is generated, it is executed immediately and the result is reflected on the local user interface. Then it is timestamped by the current value of the local state vector, stored in the local HB, and propagated to all remote sites. If the operation is *Create*, then the object it created is assigned the identifier of this operation (as specified by the COID scheme). This object is stored in the GOL and a copy is sent, along with the *Create* operation, to all remote sites. Each object in the GOL refers to the collection of operations in the HB which have been applied to it.

When a remote operation arrives and becomes causally ready for execution (refer to Definition 1.5), the collection of objects in the application scope of this operation
Figure 5.2: *Multiple versions of the rectangle are created*

are selected from the GOL by executing the *TOVER* scheme. Then, the *MOVIC* algorithm is executed to check this operation against each object in its application scope to determine the combine effect. The execution of a remote operation may create new object versions due to conflict. The new object versions (if any) are placed in the GOL according to Definition 2.11. On the user interface, object versions created due to conflict are especially marked to allow users to differentiate the objects created by concurrent *Create* operations from objects created due to conflict resolution. As shown in Figure 5.2, each of the two rectangles are marked with a 'V' character since
they were created by two conflicting operations which moved a single rectangle to two different positions. A more sophisticated display for object versions is possible, but beyond the scope of this thesis.

If the remote operation is a Create operation, then the convergence scheme needs to be applied to ensure the new object is in the same GOL position at all sites. However, instead of carrying out the undo/do/redo algorithm (in Algorithm 1.1), this operation is compared with other concurrent Create operations to determine the object ordering, and the new object is simply inserted into the appropriate position in GOL. Whenever there is a need to re-paint the user interface, objects in the GOL are drawn in their order in GOL from right to left to ensure convergent layering.

5.3 Any Undo implementation

Undo commands are generated by users from an undo table. An undo table displays a list of operations available for undo/redo as shown in Figure 5.3. The undo table is especially designed to help users select which operation is to be undone. The undo table contains the information regarding the operations (its site identifier, type, value, and status) plus the details of the object(s) (type, position, and color) that an operation is applied to. The number of columns in this table is extensible, so more information regarding operations or objects can be added later, if required.

Once an operation is selected by a user for undo/redo, an undo command containing the identifier of that operation is generated. This command will be executed locally before it is sent to all remote sites for execution. When executing an undo command, the operation identifier specified in the command is used to find the operation in HB. The TOVER scheme is used to determine the application scope of
this operation. If this operation is marked, then Redo() is invoked; otherwise Undo() is invoked. In Undo(), MOP is first called to undo the effect of the operation from objects in its scope. For each object in the scope, MOP calls UNREG to produce the inverse operation for this object before the inverse is executed. After the execution, MOVEM is invoked to remove any unrequired versions. In Redo(), VEP is first called to pre-create any required version (VEP was not integrated into MOVIC during the implementation). Then MOVIC is invoked to determine the combined effects and also to create the required versions. The actual execution of this operation is done by MOP as described above. Finally, after Undo() and Redo(), the Mark() function is called to mark or unmark the operation.

5.4 Instant locking implementation

All graphics objects in GRACE contain locking information as their attributes to indicate their locking status. The required locking information for each object are:
• \textit{lockOwner}: a list of identifiers for the sites that own the lock to this object; and

• \textit{lockStatus}: the current lock status on this object.

Whenever, a locking operation is applied to an object, the identifier of the site which generated this operation is added to the \textit{lockOwner} list of this object. On the other hand, when an unlock operation is applied to an object, the identifier of the generating site is removed from \textit{lockOwner}.

Region locks are implemented as a type of graphics object (they also contain \textit{lockOwner} and \textit{lockStatus} attributes). They are very similar to rectangle objects. A region lock object is created when a locking operation to lock a region is executed. In order to simplify the validation process, region locks are stored in the GOL and treated just like any graphics object.

There are two cases which are handled differently during the validation process. Most of the operations in GRACE, including \textit{Destroy}, and all update and locking (lock and unlock) operations, require the target object in GOL to be first selected before the operation can be generated. Validation can be achieved for these types of operations by preventing the selection of that object if the request is invalid (if an object cannot be selected, then no operation can be generated to edit that object). A request is invalid if the object which the user tries to select has one or more site identifiers in the \textit{lockOwner} list, and the local site identifier is not in this list. \textit{Create} operations are handled differently because its generation does not require the prior selection of an object. The validation for \textit{Create} is carried out after the size of the object has been determined. Then the object is compared with any region lock in GOL; if it overlaps a region lock object and the local site identifier is not in the
<table>
<thead>
<tr>
<th>Lock Status</th>
<th>Outline Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstable</td>
<td>Yellow</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Green</td>
</tr>
<tr>
<td>Shared</td>
<td>Blue</td>
</tr>
<tr>
<td>Locked by remote site</td>
<td>Red</td>
</tr>
</tbody>
</table>

Figure 5.4: *Different color outlines to represent different locking status*

lockOwner list of the region lock, then this Create request is invalid and the object is discarded.

When an object is locked by a locally generated locking operation, its lockStatus is set to unstable while it waits to become stabilized. While it is still unstable, if another operation arrives to lock the same object, then the lockStatus of this object is set to shared (the object is locked by a shared lock). If no such operation is received before the lock stabilizes, then after stabilization, the lockStatus of the object is set to exclusive. If a site receives a remote locking operation targeting an unlocked object, then after executing this locking operation, the lockStatus of this object is set to remote, which indicates this object has been locked by a remote site. However, if the object targeted by the newly arrived locking operation already has lockStatus of remote, then applying the locking operation does not change the lockStatus (only the originating site identifier of the operation is added to the lockOwner list).

The user interface plays an important role in determining whether locking is usable. How to indicate an item has been locked? How to distinguish between different locking status? In GRACE, a locked item is indicated by a thick colored outline of an object or region as shown in Figures 5.5 and 5.6. Different colored outlines are used to indicate different locking status, as shown in Figure 5.4.
Figure 5.5: *A locked object and region on site 1*

When a lock is generated, it is in its unstable period, the color of the lock at the local site is yellow. When the lock has stabilized, then the color of this lock becomes green (if it is an exclusive lock) or blue (if it is a shared lock). However, at the remote site, when a locking operation is received and executed, the color of this lock is red (owned by another user(s)). The displays of locks at two sites are illustrated in Figures 5.5 and 5.6 respectively. In this scenario, there is a region lock and an object lock. The region lock is owned by Site 1. So in Site 1 (Figure 5.5), the region lock has the color green and in Site 2 (Figure 5.5), this region lock has the color red. The
object lock is owned by Site 2, so in Site 2, this object lock has the color green, and in Site 1, this object lock has the color red.

5.5 Maintaining executed operations

Executed operations need to be kept by the system for two reasons:

1. These operations may be used for maintaining convergence and intention preservation.

2. These operations may be selected by the users for undo.
The ideal situation is that all executed operations are kept by the system in HB. However, there are limitations on the amount of available system memory. Therefore, it is not feasible to keep all executed operations. Hence, a scheme needs to be applied to removed unrequired operations from the system.

Operations in HB are used by the multiple object version scheme to maintain consistency. Therefore, operations not required by these two schemes can be removed from HB. In both schemes, a newly arrived operation, $O_{new}$, has to be checked against any operation that is independent of $O_{new}$ for possible conflict or out-of-serial-order execution. Therefore, if it has been known that for any operation $O$ in HB, and all operations independent of $O$, has arrived and been executed at that site, then $O$ is not required by these two consistency schemes. This condition is the same as in REDUCE. Hence, the garbage collection scheme [105] proposed for REDUCE can be directly used in GRACE to remove unwanted operations from HB.

Even though an operation may not be needed for consistency maintenance, it may still be selected for undo by users. Therefore, these operations still need to be kept. However, these operations should not be kept in the HB, because HB needs to be checked every time a new operation is executed. Hence, after the execution of the garbage collection scheme, these operations are removed from HB and stored in another list, called Extended History Buffer (EHB). Operations in both HB and EHB are available for users to undo, therefore they are displayed in the Undo Table.

Operations in EHB also need to be removed at some stage to release occupied memory space (since systems do not have infinite memory). Once an operation has been removed from EHB, it will not be available for undo. A consistency requirement which needs to be satisfied when removing operations from EHB is that after removal
of operations from EHB, the same set of operations exits in EHB at all sites. This requirement will ensure that once an undo command is generated at one site, it can be executed at all remote sites.

The above requirement can be satisfied by executing a check point function. This function is actually an implementation of a distributed consistent cut in a collaborative editing session (refer to Chandy and Lamport [13] for more detail). When a check point is executed, the whole system will be forced to enter a quiescent state, that is:

1. all operations generated before the checkpoint are executed at all sites;

2. document copies at all sites are converged;

3. state vectors at all sites are reset to initial (zero) values; and

4. HB and EHB at all sites are reset to empty.

The above check point function is currently implemented in GRACE. Check points are initiated by the users. Once a check point is executed, all operations generated before the check point cannot be undone. However, a more sophisticated EHB operation removal scheme can be developed so that only some (the same) operations are removed from EHB at all sites during check point. The conditions governing which operations to remove, and how this can be achieved are a possible future research topic.

5.6 Artificial network delay

GRACE was designed to work over the Internet environment. In such an environment, operations with arbitrarily complex concurrency and conflict relationships can
be naturally generated because of the possible long and non-deterministic communication delay. Before using the system in the Internet environment, it must be tested in a Local Area Network (LAN) environment, where communication latency is low and messages are automatically serializable by the underlying broadcast transmission media. Without special measures, conflicts due to concurrency rarely occur in the LAN environment and many scenarios commonly observable in the Internet environment are hardly detectable in the LAN environment. To facilitate the testing of the proposed algorithms, a communication delay mechanism is implemented into GRACE to delay any remote operation for an arbitrary period of time.

This delay mechanism is implemented on the receiver's site. The premise is to hold on to the remote operations received without executing them until the appropriate time (which is determined by the user). This is implemented by blocking the RSH thread outside of RE when delay is required, so that no remote operation can be executed. Any remote operation which arrives at this time will not be processed and is simply queued in the underlying socket. During this period, local operations can still be generated since UIH is still active. All local operations generated at this time will be independent/concurrent of all remote operations waiting for execution. When it is time for these remote operations to be executed, the RSH threads are unblocked. These threads will then execute each operation queued in the socket.

This delay mechanism has been implemented so that it is directly controllable via the user interface. A check box is provided so that, when it is checked, all operations arriving from remote sites will be held without being executed. Once this check box is unchecked, these remote operations will be executed. By means of this mechanism, it is possible to generate a group of operations with arbitrary concurrency and conflict
relationships, and to observe the system’s behavior under any particular scenario in any environment.

5.7 Usage experience

So far, usage experience with GRACE as a productive tool has been limited, due to the limited editing features provided by the system. This is despite the fact that GRACE already supports more operations than REDUCE (apparently graphics editing systems need a lot more features for it to be useful, compared to text editing systems). For graphics editors to be useful, they either need to be specialized for drawing certain types of diagrams (e.g. connected diagrams), or incorporated into other applications (e.g. word processors). Efforts are being directed towards building a more robust and useful system, which will be used by external users in real application contexts to evaluate the research results from end-users’ perspectives. However, such work is beyond the scope of this thesis.
Chapter 6

Conclusion

6.1 Summary

Real-time collaborative editing systems are groupware systems which allow multiple users to edit the same document at the same time. This dissertation has focused on maintaining consistency in real-time collaborative graphics editing systems which provide fast response time in the Internet environment and allow unconstrained editing of any part of the document. Initially, the inconsistency problems in this type of systems were examined. It was found that the three well-known inconsistency problems in collaborative text editing systems (divergence, causality violation, and intention violation) also manifest themselves in collaborative graphics editing systems. Causality violation can be solved by the existing methods of delaying the execution of out-of-causal-order operations. However, there was no consistent solution for resolving both divergence and intention violation problems caused by conflicting operations.

In this thesis, it was first proposed that a multiple object version scheme (in Chapter 2) be devised to solve the intention violation and intra-object divergence problems caused by the execution of conflicting operations. This scheme maintains
the intention preservation property while allowing unconstrained editing from multiple distributed sites. With this scheme, versions of the same object are created, one for each conflict operation. The effect of each conflicting operation is preserved on a different version. From the result produced, the users can make informed decisions on which operation’s effect they wish to keep. In addition to intention preservation, this scheme also ensures intra-object convergence without having to perform system initiated undo/do/redo on executed operations. This multiple object version scheme is the first scheme based on attribute level conflict. Compared to object level conflict, this scheme has finer granularity of conflict which leads to the advantage of minimizing the number of versions, hence reduces the overhead of version management. An object identification scheme was proposed to work in conjunction with the multiple object version scheme. This identification scheme is able to uniquely and consistently identify objects as well as being able to determine versions of the same object. Then, a comparison was made between text and graphics collaborative editing systems. This comparison discovered that the text and graphics editing systems use different methods to find objects for applying operations. Furthermore, operational transformation (used in the text domain) and the multiple object version scheme are complimentary concurrency control techniques which solve inconsistency problems of different natures. This multiple version scheme is the foundation upon which further research is built.

The multiple object version scheme is used for the execution or ‘DO’ of operations. In addition to the capability of ‘DO’, the capability of ‘UNDO’ is also an important feature of any editor. Undo can be used for error recovery and alternatives exploration. Proposed was an Any Undo scheme (in Chapter 3), which maintains
consistency under the condition that users may select any operation to undo at any
time. Any Undo is especially important in collaborative editing systems because it
can be used to support local or global undo and also multiple undo models. Our
contributions to undo in collaborative graphics editing systems include: defining the
undo effect for undoing any executed graphics operation; designing algorithms to pro-
duce the required undo effect; and designing algorithms to maintain consistency by
removing and re-creating object versions. This is the only undo scheme which sup-
ports the concurrency control technique of the multiple object version. Furthermore,
as far as known, no other undo scheme in collaborative graphics editing systems has
the flexibility of this Any Undo scheme.

In addition to the conflict resolution techniques discussed in Chapters 2 and 3,
conflict prevention techniques for consistency maintenance are also studied in this
thesis. Proposed was an optional instant locking scheme (in Chapter 4) which can
be used to prevent the generation of conflicting operations while the responsiveness
of the system is not sacrificed. This locking scheme is optional because even if locks
are not placed, syntactic consistency of the system is still maintained by the multiple
object version scheme. When locking is used semantic consistency can be preserved.
Hence, the multiple object version scheme and the optional instant locking scheme
are complimentary techniques designed to work together collaboratively. Two types
of locks were proposed, object and region locks. Object locks are used to lock objects,
and its usefulness is obvious. Region locks are used to lock editing areas. In addition
to conflict prevention, regions locks can be used for access control or as a group
awareness mechanism.

All algorithms and schemes presented in this thesis have been implemented in
the Internet-based GRACE prototype system written in the programming language Java. The current GRACE prototype system has been developed mainly to test the feasibility of our approach and to explore system design and implementation issues (discussed in chapter 5). Efforts are being directed towards building a more robust and useful system, which will be used by external users in real application contexts to evaluate the research results from the end-users’ perspectives.

6.2 Future work

Collaborative editing systems is an area with many challenging issues which need to be resolved. The work presented in this thesis can be used as a foundation for many future research directions. The following future topics have been identified for investigation:

- There are many different types of operations in graphics editing systems. The work in GRACE has focused on some of the most essential operations. Still, there are many other operations which have not yet been investigated. For example, operations to group and ungroup objects, operations to change the layering order of the objects, and operations to rotate objects. Developing these operations is important for building useful collaborative graphics editing systems. Once GRACE has enough functionality to enable users to carry out constructive work, usability study can be performed.

- Most of the work in collaborative editing systems has concentrated on text and graphics environments. There are many other types of documents where collaborative editing technologies can be applied, such as three-dimensional graphics documents, bitmap documents, spreadsheet documents [45, 81], the combined
text and graphics documents, and other multimedia (sound and video) documents, etc. Research into different document types should produce interesting and important results comparable to the work in text and graphics documents.

- Documents in collaborative editing systems can be classified as discrete interactive medium which change their states only in response to user initiated operations. This is in contrast to continuous interactive medium [68, 67] which can change their state by both user initiated operations and the passing of time. Examples of continuous interactive medium are networked computer games, virtual reality, and distributed simulations for training and education, etc. Consistency maintenance in continuous medium is relatively unexplored compared to discrete media. So some interesting results in this area can be expected.

- Being aware of other users’ locations, activities, and intentions relative to the task enables people to work together more effectively. Providing group awareness information is an important feature of any groupware. Group awareness schemes can determine the success or failure of collaborative editing systems. There are a number existing results in group awareness [26, 40, 41, 48, 49, 50]. Some of the existing group awareness mechanisms are generally applicable to real-time collaborative editing systems. They may be directly applied to GRACE. However, none of these results specifically address group awareness issues for real-time collaborative graphics editing systems with multiple version concurrency control, Any Undo, and optional locking. Hence, new group awareness mechanisms specific to GRACE need to be developed (e.g., to indicate versions of the same object, to display operation history for Any Undo etc.).
Another direction is to tailor collaborative editing systems for some specialized applications. One of the applications envisaged is to tailor collaborative editing systems into collaborative Computer Assisted Software Engineering [27] (CASE) tools. For example, GRACE can be extended so that it can be used to draw connected diagrams such as entity-relationship diagrams, dataflow diagrams, flow charts, and state-transition diagrams, etc. REDUCE can be tailored for collaborative programming [78], debugging [94] and code inspection.

Research in real-time groupware systems has drawn inspiration from traditional databases and distributed computing techniques (e.g. causal/total ordering of events, vector logical clock timestamping, locking, serialization, write update/invaidate, etc.), and has also led to the invention of non-traditional techniques (e.g. operational transformation [103], optional locking [107], the multiple object version scheme, etc.) to address its special issues, such as intention preservation. The generalization and application of these non-traditional techniques to other areas of distributed computing and CSCW, is an exciting general direction for future exploration.
Bibliography


[34] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. Australian Computer Science Communications, 10(1), February 1988.


Index

Att.Type(O), 45
Att.Value(O), 45
G{Ox}, 51
G{ }, 51
Id(G), 71
Id(O), 71
MostRecentlyOverwrittenBy(), 117
S(), 104
Target(O), 45
⇒, 39
⊙, 45, 83
⊙, 45, 83
⊙D, 79
⊙I, 82
∥, 31
→, 31
≥, 109
isMarked(), 118
mark(), 118
access control, 10
adding objects, 92
all-operation-effect, 48
Any Undo, 103
Any Undo scheme, 42
architecture, 15
centralized, 16
hybrid, 19
replicated, 18
awareness information, 9
bitmap graphics, 13
causal consistency, 24
causal ordering, 31, 36
causality preservation, 32
causality violation, 28
causally ready, 38
CGS, 55
chat systems, 5
COID, 134
COID scheme, 74
collaborative editing systems, 3, 11
asynchronous, 12
real-time, 12
synchronous, 12
collaborative graphics editing systems, 13, 25
consistency model, 30
inconsistency problems, 25
concurrency control techniques, 33
combined effect rules, 50
compatibility relationship, 83
Compatible groups set, 55
compatible groups set, 55
compatible relation, 45
compulsory locking, 144
computer conferencing, 6
concurrency control, 10
conflict, 26
conflict prevention, 88
conflict relation, 45
conflict relation matrix, 55
conflict relation triangle, 55
conflict resolution, 41, 89
consistency, 70
consistency maintenance, 20
  collaborative graphics editing systems, 25
  database systems, 21
  distributed shared memory, 23
consistency model, 32
consistency object identification, 69
consistent object identification scheme, 74
convergence, 32, 85
  inter-object, 86
  intra-object, 85
convergence scheme, 38
coordination systems, 6
CRM, 55
CRT, 55
CSCW, 1
dependent operations, 31
direct conflict, 79
direct selective undo, 139
distributed, 13
divergence, 26
  inter-object, 27
  intra-object, 26
document structure, 92
electronic meeting rooms, 5
e-mail, 4
Ensemble, 89
equivalent CGS, 57
existence, 138
explicit lock generation, 147
extended indirect conflict, 125
global undo, 103
GOL, 162
GRACE, 14
graphics objects, 14
graphics objects list, 162
graphics operations, 14
create, 15, 137
destory, 15
destroy, 137
revive, 139
update, 15, 138
  fill, 15
  move, 15
  resize, 15
graphics redo rule, 114
graphics undo rule, 111
group decision support systems, 5
group interfaces, 8
GroupDesign, 140
GroupDraw, 89
groupware, 2
  applications, 3
  asynchronous, 2
  classification, 2
  design issues, 8
  synchronous, 3
history buffer, 39, 162
identifier, 71
implicit lock generation, 147
implicit removal scheme, 118
independent operations, 31
indirect conflict, 82
instant locking scheme, 150, 152
  concurrent versions, 155
  instant lock, 152
  instant lock sharing, 153
instant messaging, 5
instant optional locking scheme, 42
intelligent agents, 6
intention, 32
intention of an operation, 31
intention preservation, 32
intention violation, 29
inter-object convergence, 86
intra-object convergence, 85
item, 146
local state vectors, 162
local undo, 103
lock ownership, 147
lock stabilization, 158
lockable item, 146
locking, 34, 88, 143
locking divergence, 149
lockOwner, 167
lockStatus, 167
Lotus Notes, 8
makeVisible(), 121
masking relation, 140
message systems, 4
MOP, 121
most recently overwritten, 112
MOVEM, 128
MOVIC, 64, 132
multi-player games, 7
multiple object version application, 121
multiple object version merging, 128
multiple object version scheme, 41, 44
NCGS, 58
NCGS specified combined effect, 61
newsgroups, 4
Normalized compatible groups set, 58
null-effect, 46

object lock, 146
object targeting systems, 96
targeting by identifier, 96
targeting by position, 96
object/vector graphics, 13
optimistic locking, 144
optional locking, 144
organization structure, 9
OVER scheme, 135
overwritten, 106, 109
PRAM consistency, 24
real-time, 12
Redo(), 136
REDUCE, 30
region lock, 146
removing objects, 92
responsive operation generation, 147
revised conflict definition, 78
semantic, 29, 32
sequential consistency, 23
serial equivalence, 22
serialization, 35, 149
session control, 10
single-operation-effect, 47
state vector, 37
strict consistency, 23
SV, 37
syntactic, 29
syntactic effect, 32
targeting by identifier, 96
targeting by position, 96
timestamp, 38
timestamp ordering, 35
Tivoli, 90, 140
total ordering relation, 39
TOVER scheme, 76
traceability, 70
transformation, 36
turn-taking, 33
unconstrained, 13
undo effect, 104
undo operation representation, 117
undo property, 104
Undo(), 136
Undo/Do/Redo Scheme, 39
undo/redo effect generation algorithm, 116
uniqueness, 70
UNREG, 116
unstable period, 151
updating objects, 95
user initiated undo, 102
valid request, 148
validation, 147
vector logical clock, 36
VEP, 131
version creation, 84
version layering property, 88
version management, 118
version pre-creation, 131
version re-creation, 129
version removal, 126
visible effect, 110
weak consistency, 24
workflow systems, 7
write-invalidate, 24
write-update, 24