IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 6, 2017, accepted September 29, 2017, date of publication October 9, 2017,
date of current version November 7, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2760914

A Novel DAL Scheme With Shared-Locking for Semantic Conflict Prevention in Unconstrained Real-Time Collaborative Programming

HONGFEI FAN[1], (Member, IEEE), HONGMING ZHU[1], QIN LIU[1], YANG SHI[1], (Member, IEEE), AND CHENGZHENG SUN[2], (Member, IEEE)

[1]School of Software Engineering, Tongji University, Shanghai 201804, China
[2]School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798

Corresponding author: Yang Shi (shiyang@tongji.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61702374, Grant 61772371, and Grant 61202382, in part by an Academic Research Grant MOE2015-T2-1-087 from MOE Singapore, in part by the Shanghai Sailing Program under Grant 17YF1420500, in part by the National Key Research and Development Program of China under Grant 2016YFB1000805, in part by the Fundamental Research Funds for the Central Universities, and in part by the Scientific Research Foundation for the Returned Overseas Chinese Scholars. This paper is an expanded version of [7] accepted and presented at the 2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD 2017), 26-28 April 2017, Wellington, New Zealand.

ABSTRACT Real-time collaborative programming allows a team of programmers to concurrently edit the shared source code document at the same time. To support semantic conflict prevention in real-time collaborative programming, a dependency-based automatic locking (DAL) approach was proposed in prior work, which automatically grants locks on source code regions with dependency relationships. The prior DAL scheme was devised under two assumptions that are not realistic, and together with other restrictions, they become serious problems in applying the DAL approach and techniques in real-world programming scenarios. To address the issues under the prior DAL scheme, this paper presents a novel DAL scheme with a shared-locking approach, which ensures the responsiveness, effectiveness, and consistency of semantic conflict prevention in unconstrained real-time collaborative programming. Under the novel DAL scheme, programmers can perform concurrent editing operations with overlapping locking scopes and perform editing operations that may dynamically change the source code structure, while three types of shared-locking are allowed under well-defined circumstances with reasonable design rationales. In addition, we have presented major technical issues and solutions in realizing the scheme, which has been implemented in a research prototype. Experimental evaluations have confirmed the good performance of the novel DAL scheme and its supporting techniques.

INDEX TERMS Dependency-based automatic locking (DAL), locking state update, real-time collaborative programming, responsiveness, semantic conflict prevention, shared-locking.

I. INTRODUCTION

Software development tasks, especially the programming work, require sophisticated collaboration among programmers. In general, there are two categories of collaborative programming approaches and supporting techniques [6]. Non-real-time collaborative programming is a traditional approach that has been widely applied in the community, which is supported by version control systems. It is considered as non-real-time collaboration because each programmer's update on the source code will be kept private until the modified source code documents are committed to the repository and further merged into other programmers' local source code copies. In contrast, real-time collaborative programming is a new approach, which supports a group of programmers to view and edit the shared source code document at the same time, while their concurrent updates can be instantly propagated to each other. During the collaboration process, each programmer can freely edit any segment of the shared source code document, and concurrent changes made by multiple programmers can be automatically integrated. Because of its benefits in improving the quality and productivity of software development, this emerging technique has attracted increasing interests from the community in the recent years [3], [4], [8], [11], [13]–[15], [17], [18], [25], [26].

The generic real-time collaborative editing technique plays an essential role in enabling real-time collaborative programming. To achieve high local responsiveness of real-time collaborative editing over the Internet where communication latency is unavoidable, a *replicated architecture* has been commonly adopted, where the document being collaboratively edited is replicated at the local workspaces of all collaborating sites [22]. With such architecture, each local editing operation issued by a user can be immediately executed on the local replica without any delay (thus being responsive), and then the operation will be instantly propagated to all other collaborating sites for remote execution (thus achieving real-time propagation and merging). Consequently, there arises a fundamental requirement: all distributed copies of the document should be identical after all editing operations have been propagated and executed remotely. This requirement is commonly regarded as the *syntactic consistency maintenance* of the shared document, which is a classic problem in the community of CSCW (computer-supported cooperative work) [21]. The sophisticated *operational transformation* (OT) techniques have been invented, developed and widely applied [21], [22], which are dedicated for supporting syntactic consistency maintenance in a variety of real-time collaborative editing systems, including the real-time collaborative programming environment in this study.

In contrast, there exists another category of consistency issues, named *semantic consistency*, which is concerned with whether the shared document content makes sense in specific domains [21], [23]. Particularly, in the domain of real-time collaborative programming, semantic consistency is concerned with *whether the shared and syntactically-consistent source code content is correct with respect to problem-solving logic*. Our prior work proposed a *dependency-based automatic locking* (DAL) approach for supporting semantic consistency maintenance (also regarded as semantic conflict prevention) in real-time collaborative programming [5], and implemented a prototype which confirmed the feasibility of the DAL approach [4]. However, the prior DAL scheme was devised under two assumptions, which are not realistic in real-world programming scenarios. Without the assumptions, the scheme may work incorrectly under certain circumstances. Together with other restrictions, the problems are serious obstacles in applying the DAL approach and techniques in real-world software development processes. In this paper, we contribute a novel DAL scheme with a shared-locking approach, as well as a set of supporting techniques for implementing the scheme, which addresses the issues in the prior work.

The rest of this paper is organized as follows. Firstly, we briefly review our prior work on semantic conflict prevention with the basic DAL scheme in Section II. Secondly, we provide detailed analysis of the problems under the prior DAL scheme in Section III. Thirdly, we propose a novel shared-locking approach for addressing the issues, and discuss the design rationales in Section IV. Consequently, with the shared-locking approach, we present major technical issues and solutions for implementing the novel DAL scheme in Section V. Following that, we present the prototype implementation and a comprehensive set of performance evaluations in Section VI. Finally, we compare this work with related techniques and studies in Section VII, and summarize our contributions in Section VIII.

II. REVIEW OF PRIOR WORK ON DEPENDENCY-BASED AUTOMATIC LOCKING (DAL)

A. FUNDAMENTALS

Based on investigations, it was observed that during a real-time collaborative programming session, semantic conflicts may occur when multiple programmers are performing *concurrent editing operations* in the same source code region or in multiple source code regions with *dependency relationships* [5]. In the context of object-oriented programming (e.g., programming with Java), a *source code region* refers to a *method* or a *field* within a class, and a *dependency relationship* refers to a *method-field-reference* (i.e., a reference from a method to a field) or a *method-method-invocation* (i.e., an invocation from one method to another method).

For example, when two programmers P_1 and P_2 are concurrently editing the same Java source code document (the *Stack* implementation) presented in the left part of Fig. 1, semantic conflicts may occur if both of them are concurrently editing the *push* method, which is a self-contained source code region. In addition, even if they are editing different source code regions, semantic conflicts may also occur when their working regions have dependency relationships (e.g., P_1 is editing the *popList* method which invokes the *pop* method, while P_2 is exactly editing the *pop* method).

For the convenience of discussion, several terms had been defined as follows [5]:

1. Given a source code document, a *basic region* refers to a piece of source code content which forms a semantically meaningful and self-contained unit. In contrast, an *open area* refers to a piece of source code content outside all basic regions. For example, given the source code document in Fig. 1, there are 9 basic regions and 8 open areas.
2. For any two basic regions A and B, if A depends on B in terms of *semantics*, then there is a *dependency relationship* from A to B, denoted as $A \rightarrow B$, and B is called a *depended region* of A. Given two basic regions A and B, if neither $A \rightarrow B$ nor $B \rightarrow A$, then A and B are independent. Dependency relationship is *transitive*: given three basic regions A, B and C, if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.
3. A *dependency graph* (DG) is a directed graph, in which: (a) a *node* represents a *basic region* in the source code; and (b) an *edge* from node A to node B represents a dependency relationship $A \rightarrow B$. For example, the right part of Fig. 1 illustrates the DG in accordance with the source code.

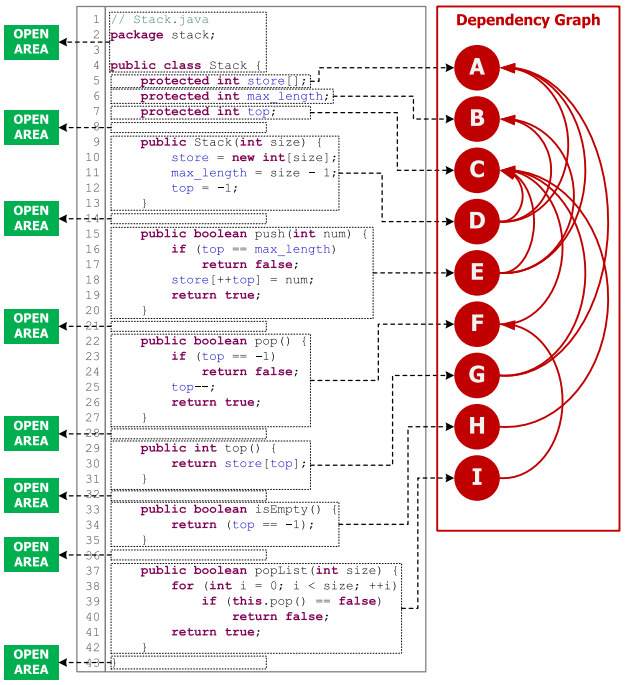

FIGURE 1. Basic regions, dependency relationships, and dependency graph.

B. DEPENDENCY-BASED AUTOMATIC LOCKING (DAL)

It is necessary to support semantic conflict prevention in real-time collaborative programming. For preserving maximum collaboration concurrency while preventing semantic conflicts, we proposed to allow concurrent programming work in the same source code document as long as *such concurrent work does not appear in the same basic region or in multiple basic regions with dependency relationships* [5]. With this principle, a programmer will not be allowed to edit a source code region until requesting and obtaining a lock on the region to be edited (denoted as *working region*) and locks on other source code regions that the working region depends on (denoted as *depended regions*). Furthermore, to avoid programmers' manual effort in requesting and releasing locks, such locking mechanism is preferable to work automatically.

Based on abovementioned ideas, we proposed a *dependency-based automatic locking* (DAL) approach in prior work [5]: whenever a programmer attempts to issue an editing operation on the source code document, the system automatically detects the working region (if any) and derives its depended regions, grants locks on the working region and depended regions for the programmer,[1] and releases the locks after the programming work in the working region is completed. Accordingly, under the basic DAL scheme, whenever a local editing operation is issued by the programmer, a *permission check* procedure will be executed, which grants or denies the editing permission according to the following permission check conditions [5]:

[1]If the programmer's editing operation falls in an open area of the source code, no lock is required.

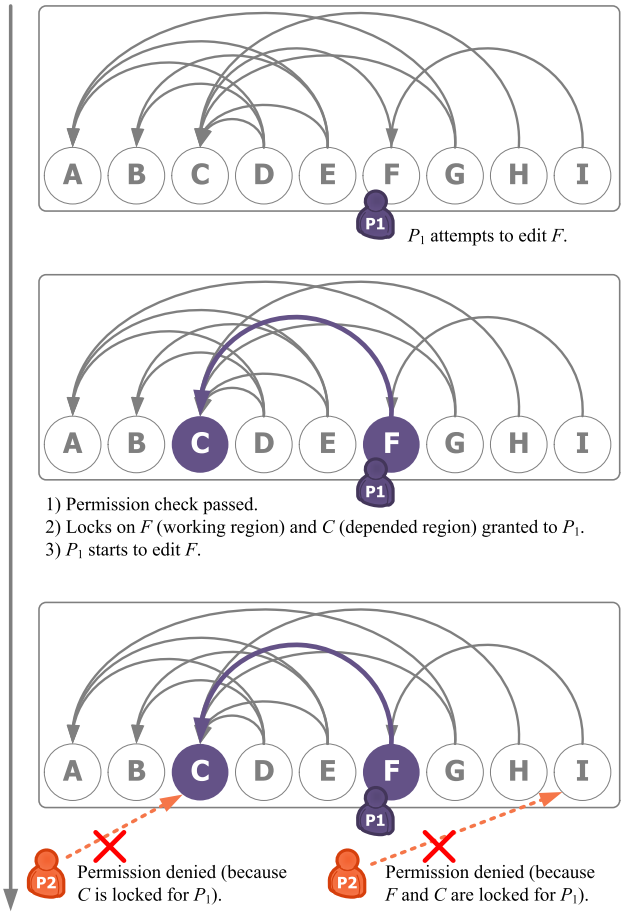

FIGURE 2. Simple illustration of the DAL permission check mechanism.

Definition 1: Permission Check Conditions under the Basic DAL Scheme

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

a) O's position falls in an *open area*; or

b) O's position falls in a *basic region* W, and:

i) W is currently locked by this programmer as a *working region*; or

ii) Neither W nor W's *depended region* is locked by any other collaborating programmer.

Fig. 2 illustrates the permission check mechanism, where the DG originates from Fig. 1. Initially, when programmer P_1 attempts to edit region F (which refers to the *pop* method in Fig. 1), the editing is permitted because both the targeted working region and its depended region (region C) are free at the moment. Accordingly, a DAL locking state update procedure is executed, which locks regions F and C for P_1, and consequently, no other programmer can edit any of them. For example, when another programmer P_2 attempts to edit region C, the editing will be denied. In addition, if P_2 attempts to edit region I, although it is not locked, the editing would also be denied because its depended regions F and C are currently locked for P_1. Based on the DAL principle, locks are always granted and released as a group (covering the working region and depended regions). In this case, P_2 cannot obtain

the entire group of required locks (on regions I, F and C) at the moment, and therefore need to wait until those locks held by P_1 are released.

It is worth mentioning that the prior DAL scheme was devised under the following two assumptions:

a) When concurrent editing operations are issued by multiple programmers, their corresponding locking scopes never overlap on any source code region; and
b) During the programming process, the DG of the source code remains *static*, i.e., editing operations never create or delete basic regions and dependency relationships.

Obviously, the two assumptions are not realistic in practice. Without the assumptions, locking operations may conflict with each other, and the DAL scheme may work incorrectly under various circumstances. Together with other restrictions, they are serious problems and obstacles in applying the DAL approach and techniques in real-world programming scenarios. We firstly analyze the problems in Section III, and then present the approach and solutions in Sections IV and V.

III. MAJOR PROBLEMS OF THE PRIOR DAL SCHEME

A. RESTRICTION IN SUPPORTING CONCURRENT WORK ON INDEPENDENT WORKING REGIONS WITH COMMON DEPENDED REGIONS

The prior DAL scheme grants *exclusive locks* on the working region and its depended regions with respect to a programmer's editing operation, which is overly restrictive but useless for semantic conflict prevention. To illustrate the problem, consider the simple example in Fig. 3 below.[2] In this case, basic regions A and B are independent of each other, but they share common depended regions C and D. Suppose that a programmer P_1 firstly started to work in region A, and obtained locks on the working region A and its depended regions C and D. Consequently, another programmer P_2 attempts to work in region B. According to the general conditions of semantic conflicts [5], concurrent work on regions A and B will not lead to semantic conflicts because they are independent of each other. However, the prior DAL scheme prohibits such concurrent work, because the system justifies that P_2 should obtain exclusive locks on the targeted working region B and its depended regions C and D, but part of them (C and D) are currently locked for P_1.

This problem is serious in real-world programming scenarios, because it is common that many class fields and utility methods can be referenced and invoked by multiple methods, which are independent of each other. Under the prior DAL scheme, when one programmer is editing one of those methods, no one else will be allowed to concurrently edit another one.

[2]In this paper, locks granted on source code regions are represented by labels below the DG nodes. A lock granted to the programmer P_x on a basic region R is represented by a label P_x written below the node R in the DG. Furthermore, if there is a superscript W placed beside P_x, it indicates that R is a *working region* of P_x; otherwise, it is a *depended region* of P_x.

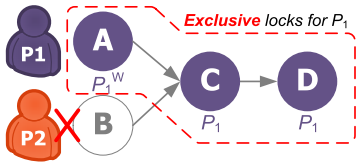

FIGURE 3. Restriction in supporting concurrent work on independent working regions with common depended regions.

B. PROBLEM WITH CONCURRENT EDITING OPERATIONS WITH OVERLAPPING LOCKING SCOPES

With the *replicated architecture* [5], the source code and the DAL locking state data are replicated at each collaborating site. This ensures the responsiveness of permission check: whenever a programmer issues an editing operation, the permission check procedure simply consults the local locking state data only, and immediately grants or denies the permission (without blocking the programmer). In addition, under the prior DAL scheme, whenever a locking operation (i.e., *grant* or *release* locks) is triggered, it will be attached to the editing operation, propagated to remote sites, and executed together with the editing operation for producing consistent locking states over all sites.

However, concurrent locking operations (which are attached to concurrent editing operations) may conflict with each other and produce inconsistent locking states in certain scenarios. When multiple programmers concurrently start to edit the same non-locked region or multiple non-locked regions with dependency relationships and/or common depended regions, all of them can be granted the editing permissions based on their local locking state replicas, but their locking scopes are overlapping. In other words, the same source code region will be locked by different programmers at different sites, leading to inconsistency problems.

For example in Fig. 4, when two programmers P_1 and P_2 concurrently start to edit the same region A, both of them will be granted the editing permissions, because each permission check is solely based on the local locking state, and their targeted working region and depended regions are all non-locked at both sites at the moment of the permission check. However, when their editing operations and the attached locking operations arrive at the remote sites, a conflict is produced: when P_2's editing operation (together with the attached locking operation) arrives at P_1's site, the execution of the locking operation fails because the targeted regions A, B and C have already been locked for P_1; similarly, when P_1's editing operation arrives at P_2's site, the execution of the locking operation also fails due to the same reason. Eventually, the locking states at the two sites are inconsistent.

Fig. 5 illustrates another example, where programmers P_1 and P_2 concurrently start to edit two different source code regions $A$and B with a dependency relationship $A \rightarrow B$. Similarly, due to the overlapping locking scopes of the two concurrent locking operations (i.e., locking regions $\{A, B, C\}$ for P_1 and locking regions $\{B, C\}$ for P_2), a locking operation

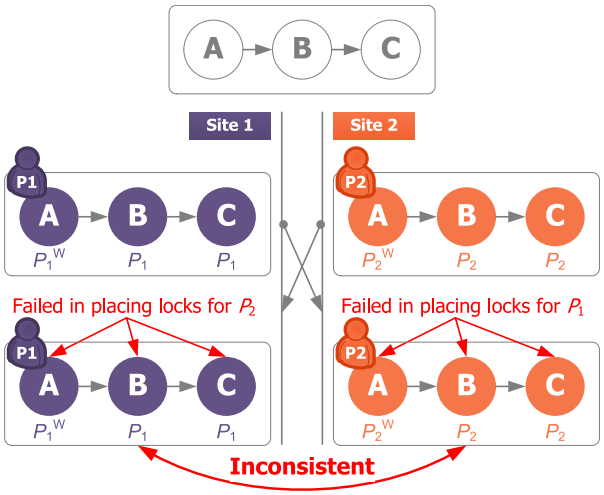

FIGURE 4. Locking operation conflict caused by concurrent editing operations on the same source code region.

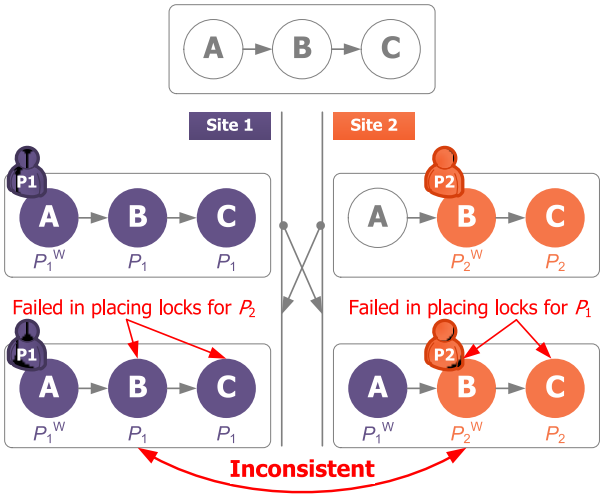

FIGURE 5. Locking operation conflict caused by concurrent editing operations on multiple source code regions with dependency relationships.

conflict is produced, which results in inconsistent locking states.

C. PROBLEM WITH DYNAMIC DG EDITING OPERATIONS

As aforementioned, the prior DAL scheme assumes that the DG (i.e., source code structure) remains static during the programming process, which is obviously unrealistic. In real-world programming scenarios, editing operations on the source code document may affect the DG from time to time. An editing operation may create or delete a method (e.g., see Fig. 6 (a)), create or delete a method invocation (e.g., see Fig. 6 (b)), and create or delete a reference from a method to a field (e.g., see Fig. 6 (c)). In general, there are two types of dynamic DG changes: (a) creation or deletion of a basic region; and (b) creation or deletion of a dependency relationship. In the rest of this paper, the term *dynamic DG editing operation* refers to an editing operation with an effect of changing the DG, whereas the term *static DG editing*

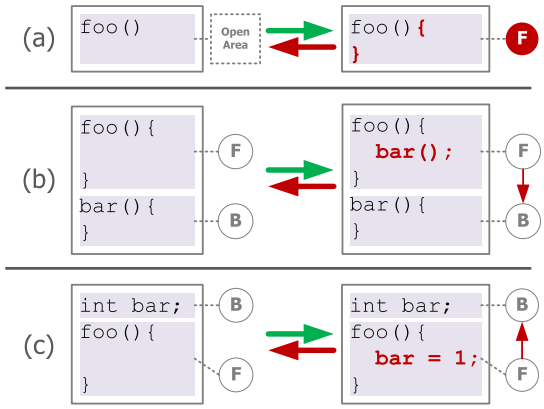

FIGURE 6. Examples of dynamic DG editing operations.

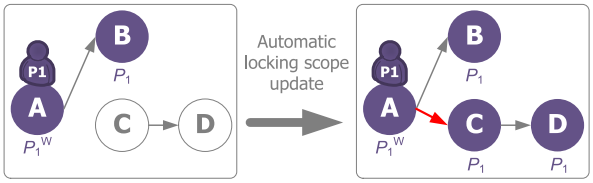

FIGURE 7. Automatic locking scope update in the face of dynamic DG editing operations.

operation refers to an editing operation without such effect.

In recognizing the existence of dynamic DG editing operations, additional actions are necessary for preserving the effectiveness of semantic conflict prevention. Following the general principle of the DAL approach, it is necessary and reasonable to dynamically update a programmer's locking scope in the face of dynamic DG editing operation, and to ensure that the locks granted to the programmer always correctly cover the working region and depended regions based on the latest source code content. For example, as illustrated in Fig. 7, when programmer P_1 initially works in region A and later creates a method invocation to region C, the DAL scheme should automatically update the programmer's locking scope from regions $\{A, B\}$ to regions $\{A, B, C, D\}$, so as to reflect the new dependency relationship $A \rightarrow C$.

However, such dynamic locking scope update mechanism may produce locking operation conflicts in certain scenarios. For example, as illustrated in Fig. 8, two programmers P_1 and P_2 are initially editing regions A and C respectively, with corresponding locks granted (i.e., locks on $\{A, B\}$ granted for P_1 and locks on $\{C, D\}$ granted for P_2). Later, P_1 creates a new dependency relationship $A \rightarrow C$, and based on the abovementioned locking scope update mechanism, P_1's locking scope should be updated to $\{A, B, C, D\}$, which overlaps with P_2's locking scope $\{C, D\}$, resulting in the failure of the locking operation, which further leads to incorrect locking state under the DAL scheme.

Furthermore, the scenario may become more complex when dynamic DG editing operations coexist with concurrent editing operations. For example in Fig. 9, in the beginning,

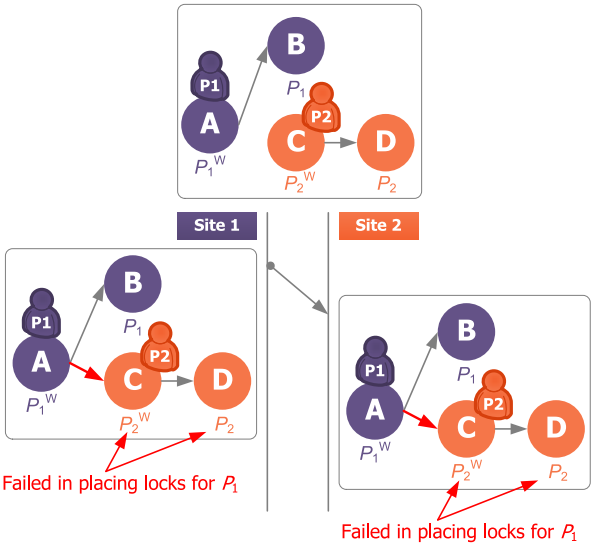

FIGURE 8. Locking operation conflict caused by dynamic DG editing operations.

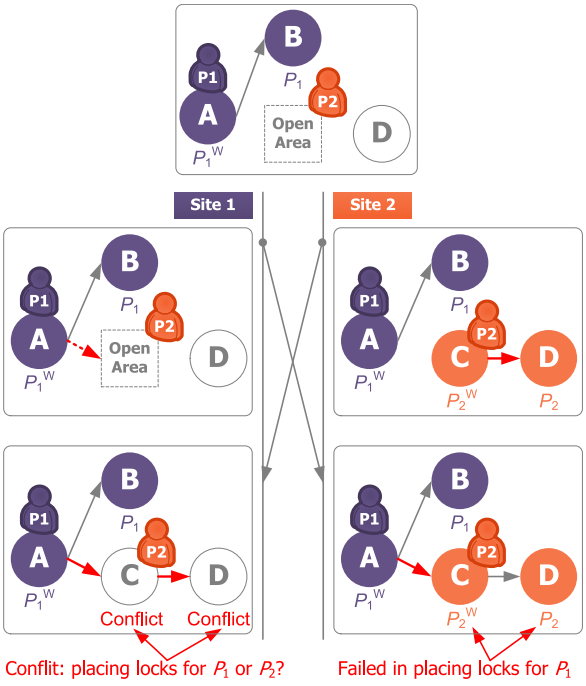

FIGURE 9. Locking operation conflict caused by dynamic DG editing operations in combination with concurrent editing operations.

programmer P_1 is working in region A with locks granted on $\{A, B\}$, while programmer P_2 is working in an open area with no lock needed. At one moment, P_1 creates a method invocation from region A to an intended method (which is still incomplete at the moment), while at the same time, P_2 completes the intended method, which transforms the open area into a valid source code region C and creates a new dependency relationship $C \rightarrow D$ simultaneously. After the two concurrent editing operations are propagated and executed at remote sites, conflicts are produced: at P_1's site, P_1's locking scope should be updated to $\{A, B, C, D\}$ and P_2's locking scope should be updated to $\{C, D\}$, resulting in locking operation conflicts on regions C and D; while at P_2's site, the locking scope update for P_1 fails because regions C and D have already been locked for P_2.

IV. THE SHARED-LOCKING APPROACH

To address the issues under the prior DAL scheme, we propose a novel DAL scheme with a *shared-locking* approach as the cornerstone, which allows several programmers to share locks on certain source code regions under well-defined circumstances. The shared-locking approach ensures the responsiveness, effectiveness and consistency of semantic conflict prevention in unconstrained real-time collaborative programming, and the three problems under the prior DAL scheme can be solved altogether with reasonable design rationales. Concretely, there are three types of shared-locking allowed under the novel DAL scheme, as presented below.

A. SHARED-LOCKING ON COMMON DEPENDED REGIONS

The novel DAL scheme allows *shared-locking* on *common depended regions* at any time. As illustrated in Fig. 10 (which reuses the case in Fig. 3), with the shared-locking approach, when P_2 starts to edit source code region B, the programmer will be granted the editing permission on the targeted region (which is independent of P_1's working region A), and obtain an *exclusive-lock* on the working region B and *shared-locks* on depended regions C and D (which are also locked by P_1 as depended regions).

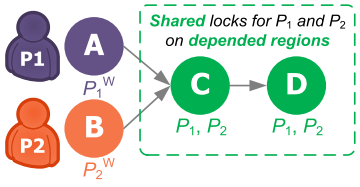

FIGURE 10. Shared-locking on common depended regions.

It is worth mentioning that, according to the general conditions of semantic conflicts [5], concurrent work on independent working regions will not result in semantic conflicts, regardless of whether they have common depended regions or not. By allowing shared-locking on common depended regions, we can greatly increase the concurrency of collaboration without sacrificing the effectiveness of semantic conflict prevention.

Accordingly, to support shared-locking on common depended regions, the *DAL permission check conditions* have been redefined as follows:

Definition 2: Permission Check Conditions under the Novel DAL Scheme

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

a) O's position falls in an *open area*; or
b) O's position falls in a *basic region* W, and:
   i) W is currently locked by this programmer as a *working region*; or

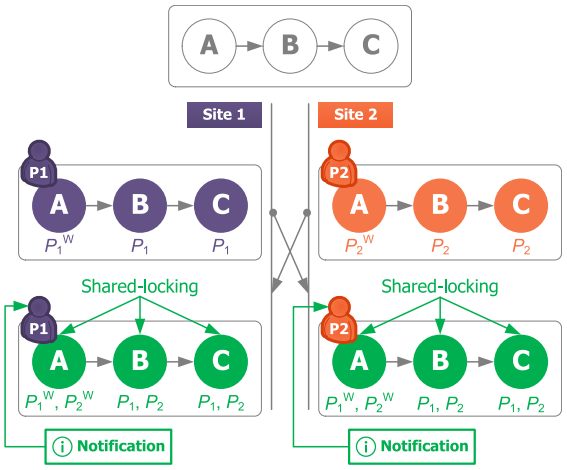

FIGURE 11. Shared-locking for concurrent editing operations with overlapping locking scopes (W-W Sharing).

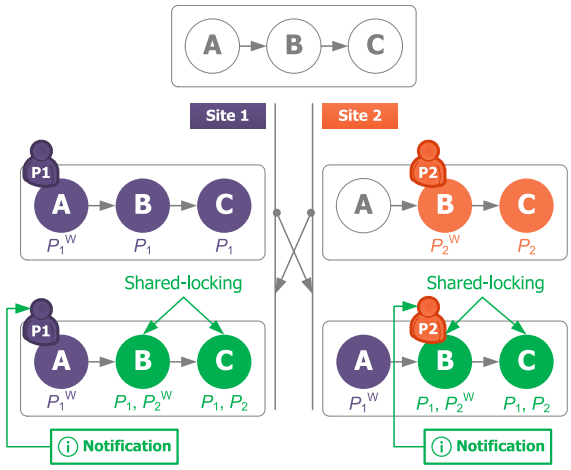

FIGURE 12. Shared-locking for concurrent editing operations with overlapping locking scopes (W-D Sharing).

ii) W is not locked by any other collaborating programmer, and none of W's *depended regions* is locked as a *working region* by any other collaborating programmer.

In the above definition, the condition b(ii) ensures that during the permission check procedure, shared-locking is allowed on depended regions only (i.e., no shared-locking on working regions is allowed).

B. SHARED-LOCKING FOR CONCURRENT EDITING OPERATIONS WITH OVERLAPPING LOCKING SCOPES

The novel DAL scheme allows *shared-locking* on *overlapping locking scopes* caused by *concurrent* editing operations. Concretely, shared-locking is allowed on working regions[3] under the following circumstances:

1. When multiple programmers concurrently start to edit *the same non-locked region*, they will be granted shared-locks on the *same (overlapping) working region*, and this type of shared-locking is denoted as *W-W Sharing*. For instance, Fig. 11 (which reuses the case in Fig. 4) illustrates the shared-locks granted on the overlapping working region A.
2. When multiple programmers concurrently start to edit *multiple non-locked regions with dependency relationships*, they will be granted shared-locks on the *overlapping working/depended region*, and this type of shared-locking is denoted as *W-D Sharing*. For instance, Fig. 12 (which reuses the case in Fig. 5) illustrates the shared-locks granted on the overlapping working/depended region B (i.e., region B is a working region for P_2 and a depended region for P_1 at the same time).

In addition, whenever a shared-lock is granted under the above circumstances, programmers involved will receive notification messages about the situation (i.e., who is/are sharing the lock on which source code region).

[3]Shared-locking on depended regions has been allowed and justified in the previous subsection, and thus will not be explicitly mentioned in the rest of the paper.

C. SHARED-LOCKING FOR DYNAMIC DG EDITING OPERATIONS

The novel DAL scheme allows *shared-locking* on *overlapping locking scopes* caused by *dynamic* DG editing operations. Supported by the shared-locking approach, any collaborating programmer, once obtained the initial editing permission granted by the DAL scheme, will be able to work continuously in the face of dynamic DG changes, without any interruption.

For example, in the case presented in Fig. 13 (which reuses the case in Fig. 8), programmer P_1 is able to work continuously in region A, even after P_1's locking scope has been automatically expanded for covering the new depended region C (which is locked for P_2 as a working region at the moment). Meanwhile, from P_2's perspective, although P_1's new locking scope has covered region C (which is currently a working region locked for P_2), P_2 can continue to work in region C.

Similarly, in another case illustrated in Fig. 14 (which reuses the case in Fig. 9), both programmers P_1 and P_2 can continuously work in regions A and C respectively, with shared-locks on the overlapping regions C and D. This case is more complex than the previous one because the shared-locking is caused by a combination of concurrent editing operations and dynamic DG editing operations. However, it is not necessary for the programmers to differentiate the cause of shared-locking, and the only important issue is to ensure that programmers are aware of the shared-locking so that they can take actions accordingly, by means of the supplementary notification mechanism.

D. DESIGN RATIONALES

The shared-locking approach has been proposed with the following design rationales, covering four aspects.

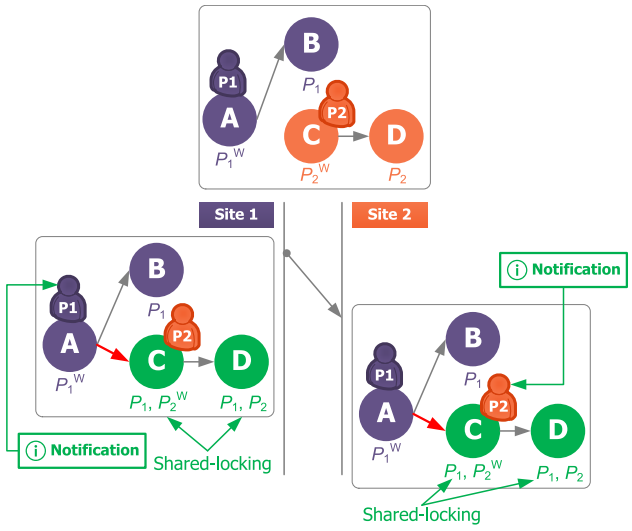

FIGURE 13. Shared-locking for dynamic DG editing operations.

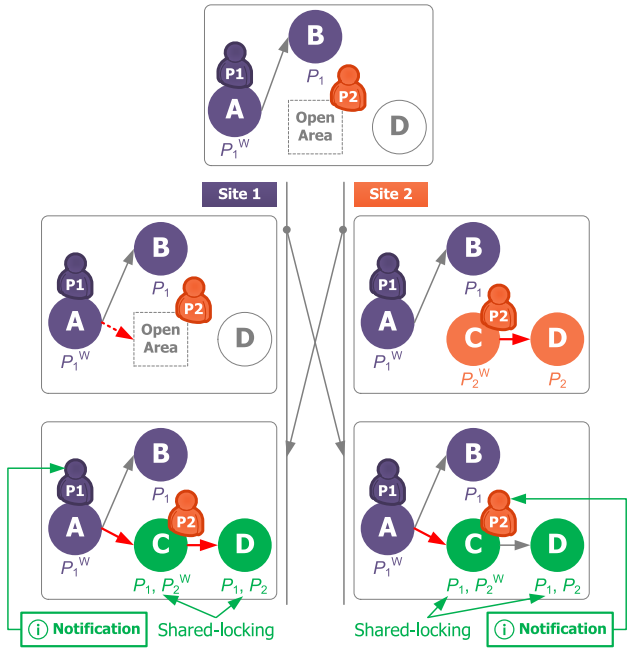

FIGURE 14. Shared-locking for dynamic DG editing operations in combination with concurrent editing operations.

Firstly and most importantly, the shared-locking approach is capable of achieving high responsiveness of semantic conflict prevention while ensuring consistent locking state maintenance over multiple collaborating sites. With the shared-locking approach, all DAL locking mechanisms can be completely distributed and localized. At every collaborating site, whenever an editing operation is issued by the local programmer, the DAL permission check can be quickly performed because it is solely based on the local locking state data, which ensures that the editing permission can be immediately granted or denied without blocking the programmer on the user interface. Whenever an editing permission is granted at a local site, the same permission can be guaranteed at all remote sites due to the adoption of shared-locking for concurrent editing operations. The locking mechanism at a remote site can quickly derive and grant locks based on the arriving editing operation, without extra communication. Consequently, there is no possibility of deadlock under the shared-locking approach. To summarize, the fully distributed and localized execution of permission check and locking state update procedures imply that there incurs no need of additional inter-site communication for any locking mechanism, which ensures the high responsiveness on the one hand, and minimizes overhead on system resource and communication bandwidth on the other hand. In contrast, if an exclusive-locking approach is adopted, additional protocols and techniques are necessary for electing a winner whenever a locking operation conflict occurs among several sites. In those scenarios, several rounds of communications may be needed between distributed sites, and the local responsiveness and user experience could be sacrificed.

Secondly, the shared-locking approach preserves continuous work for all programmers under all circumstances. Concretely, if a programmer initially obtains the editing permission on a source code region and keeps working in that region without switching to another region or open area, the editing permission will never be withdrawn. With the shared-locking for concurrent editing operations, if a programmer initially obtains the editing permission in a region (granted by the local permission check procedure), the locks will not be withdrawn if the same working region or its depended region is locked by a concurrent editing operation issued from a remote programmer. Similarly, with the shared-locking for dynamic DG editing operations, if a programmer initially obtains the editing permission in a region and keeps working in the region, the granted locks will not be withdrawn even if the programmer's locking scope has dynamically expanded and covered a working region locked by another programmer.

Thirdly, the shared-locking approach strictly complies with an essential principle of the DAL approach: if a programmer has been permitted to work in a source code region, the DAL system must always preserve correct locks on the latest working region and its depended regions for this programmer under all circumstances, which ensures the correctness and effectiveness of semantic conflict prevention.

Last but not least, in consideration of the nature of real-time collaborative programming, the shared-locking approach with the notification mechanism is most suitable in such interactive environment, because locks here are used to coordinate behaviors among human users. This nature is significantly different from that in database and distributed computing systems, where locks are used to coordinate processes for prohibiting concurrent updates on shared objects. In this study, if we apply an exclusive-locking approach alternatively, it is true that the system is able to elect a lock winner (based on predefined rules and protocols) and maintain locking state consistency when a locking operation conflict occurs. But, is

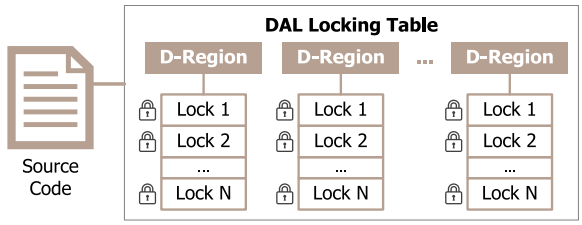

FIGURE 15. DAL locking table.

such machine-decided result the best from the collaborating programmers' perspective? In fact, when a locking operation conflict occurs, only the involved programmers have the knowledge for deciding the best action to take. This is why a notification mechanism has been integrated, which ensures the awareness. Whenever a shared-lock is granted, all programmers involved will be notified in the user interface, and they can negotiate and take actions accordingly: (a) if they judge that the concurrent editing with shared-locking is risky, they can collectively decide who should be given the exclusive permission at the moment; and (b) they may also continue the concurrent editing with shared-locking if they are able to take care of the issue and manage potential conflicts.

V. TECHNICAL ISSUES AND SOLUTIONS

With the shared-locking approach, the novel DAL scheme can be designed and implemented accordingly. There are two essential technical issues: (a) designing an appropriate locking state data structure for supporting the novel DAL scheme with shared-locking; and (b) implementing efficient and consistent permission check and locking state update procedures that fully comply with the shared-locking rules under defined circumstances.

A. LOCKING STATE DATA STRUCTURE

At each collaborating site, the DAL *locking state* is the essential information for supporting all locking mechanisms, which records the locks that are currently granted on source code regions for certain programmers. The locking state data will be retrieved by the permission check, and be dynamically updated whenever locks are granted or released.

For accommodating multiple locks granted on single source code regions under the novel DAL scheme, the locking state data structure has been designed, as sketched in Fig. 15. At every collaborating site, the source code document is associated with a *locking table*, which contains a set of *D-Regions*, and each *D-Region* corresponds to a group of locks which are granted to multiple programmers.

Within the locking table, a *D-Region* refers to a basic region (i.e., method or field) that is currently locked by one or multiple programmers, represented as <*Source Pointer*, *Lock Group*>, where the *Source Pointer* technically relates the *D-Region* to the corresponding method or field in the source code, and the *Lock Group* is a list that stores all locks that are currently granted on this region. Each lock is

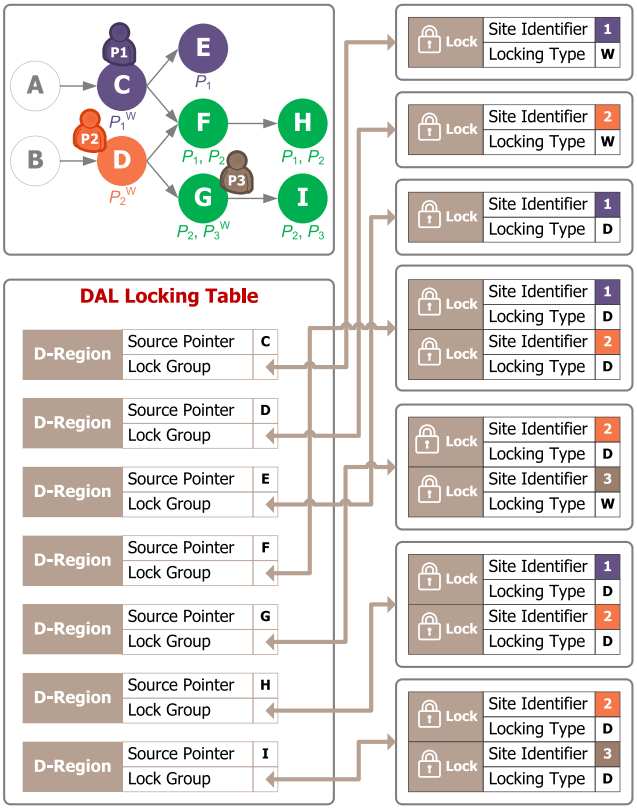

FIGURE 16. Instance of the DAL locking table.

represented as <*Site Identifier*, *Locking Type*>, where the *Site Identifier* indicates the site (i.e., programmer) that owns this lock, and the *Locking Type* indicates whether the region is locked as a working region or depended region with respect to the owner (indicated as *W* or *D* respectively). It is clear that each *D-Region* can accommodate multiple locks, which supports the shared-locking approach.

An instance of the DAL locking table is presented in Fig. 16. In this case, three programmers P_1, P_2 and P_3 are concurrently editing source code regions *C*, *D* and *G* respectively in the same source code document. In particular, regions *F*, *G*, *H* and *I* are currently shared-locked by multiple programmers.

B. UTILITY FUNCTIONS

To support essential procedures of the novel DAL scheme, the following utility functions have been designed and implemented:

1. *DetectTargetedRegion*(*O*): to detect the targeted region (if any) of the given editing operation *O*. This function returns a *Source Pointer* (which refers to a particular method or field) if *O*'s position is located within a valid source code region; otherwise, it returns *OA* if *O*'s position is located within an open area of the source code.
2. *DeriveDependedRegionSet*(*R*): to retrieve the depended regions (as a set) with respect to the given source code

region R. In particular, if R has no depended region, an empty set will be returned.

3. *ExamineLockingState*(R, P, S, T): to read the local locking table and examine whether the specified region R is currently locked by the programmer P (specified as $S = TRUE$) or any other programmer (specified as $S = FALSE$) as a working region (specified as $T = WORKING$) or depended region (specified as $T = DEPENDED$), or regardless of the region type (specified as $T = ANY$). This function returns *TRUE* or *FALSE* to indicate the result of examination. For example, suppose that the local programmer's identifier is 5, and we may invoke *ExamineLockingState*(A, 5, *FALSE*, *DEPENDED*) to examine whether source code region A is currently locked by others as a depended region.
4. *GrantLocks*(P, W, D): to grant locks for the programmer P on the working region W and the set of depended regions D, and update the local locking table accordingly.
5. *ReleaseLocks*(P): to release locks held by the programmer P, and update the local locking table accordingly.

The *DetectTargetedRegion*(O) function firstly retrieves the list of basic regions (i.e., fields and methods) in the source code document, while the retrieved information contains the *starting position* and *length* of each region. It then compares the positional properties of O with those of each retrieved region, and determines whether O falls in any basic region or not. In terms of the efficiency, the complexity of this function is $O(N)$ where N is the amount of basic regions within the source code document.

The *DeriveDependedRegionSet*(R) function starts from an empty *depended region set* (denoted as *DRS*). If R is a field (which has no depended region), then an empty *DRS* will be returned. Otherwise, if R is a method, then the sub-elements within the method body (i.e., sub-nodes of the *AST* node[4] that represents the method) will be visited, and field references and method invocations will be retrieved. If any referenced field or invoked method appears in the current source code document, then it will be added into the *DRS* (if it has not been added into the *DRS* yet). Furthermore, if the newly added depended region is a method, then the above steps will be repeated to further derive the depended regions of this method. This procedure is recursively applied until no more region can be added into the *DRS*. In terms of the efficiency, in the worst case, the specified region could depend on all other methods of the same class, and all dependency relationships (i.e., field references and method invocations) may be visited and analyzed. For each dependency relationship being visited, the function firstly checks whether the referenced field or invoked method is a basic region of the current source code document, and if so, further checks whether it already exists in the *DRS*. Accordingly, the complexity of this function is $O(NM)$ where N is the amount of basic regions and M is the amount of field references and method invocations.

The *ExamineLockingState*(R, P, S, T) function visits the local locking table and searches for the particular *D-Region* entry that is related to the source code region R by comparing R with the *Source Pointer* of each *D-Region*. Once the particular *D-Region* entry is found, the locks in the *Lock Group* of this *D-Region* are further visited and examined to check the locking state with respect to the specified conditions. In terms of the efficiency, in the worst case, all basic regions are locked and involved in the locking table, and they may be shared-locked by all collaborating programmers in the session. Accordingly, the complexity of this function is $O(N + P)$ where N is the amount of basic regions, and P is the amount of collaborating programmers.

The *GrantLocks*(P, W, D) function inserts new entries (i.e., locks) into the locking table for the specified working region W and depended region set D. For processing each region among them, the function firstly searches for the corresponding *D-Region* in the table. If the particular *D-Region* is found, a lock will be inserted into its *Lock Group*; otherwise, if no corresponding *D-Region* exists, a new *D-Region* will firstly be inserted into the locking table, and then a lock will be inserted into the *Lock Group* of this new *D-Region*. In terms of the efficiency, the complexity of this function is $O((1 + |D|) * K)$ where K is the amount of *D-Regions* within the locking table. In the worst case, $N = 1 + |D|$ and $K = N$ where N is the amount of basic regions in the source code. Accordingly, the complexity of this function can be derived as $O(N^2)$.

The *ReleaseLocks*(P) function simply visits all locks within the *Lock Groups* of all *D-Regions*. For each lock being processed, the function firstly checks whether its *Site Identifier* matches the specified programmer P; and if so, this lock will be removed from the *D-Region*'s *Lock Group*. In terms of the efficiency, in the worst case, the amount of *D-Regions* equals to the amount of basic regions in the source code (denoted as N), and the amount of locks within the *Lock Group* of each *D-Region* equals to the amount of collaborating programmers (denoted as P). Accordingly, the complexity of this function is $O(NP)$.

In addition to the preliminary complexity analysis, detailed performance evaluations on the five utility functions will be presented in Section VI-B.

C. PERMISSION CHECK PROCEDURE

Permission check is essential for ensuring the effectiveness of semantic conflict prevention. According to the permission check conditions defined in Section IV-A and the locking state data structure designed in Section V-A, the *permission check procedure* has been designed and implemented for the novel DAL scheme. Algorithm 1 implements the procedure *CheckPermission*(O, P), which determines the permission (i.e., either *PERMIT* or *REJECT*) for an editing operation O issued by the local programmer P.

[4]AST: abstract syntax tree

Algorithm 1 CheckPermission(O, P)

```
1  W := DetectTargetedRegion(O);
2  if W = OA
3    return PERMIT;
4  if ExamineLockingState(W, P, TRUE,
   WORKING) = TRUE
5    return PERMIT;
6  if ExamineLockingState(W, P, FALSE,
   ANY) = TRUE
7    return REJECT;
8  DRS := DeriveDependedRegionSet(W);
9  for each region D in DRS {
10   if ExamineLockingState(D, P,
     FALSE, WORKING) = TRUE
11     return REJECT;
12 }
13 return PERMIT;
```

As presented above, the procedure invokes the *DetectTargetedRegion*, *DeriveDependedRegionSet* and *ExamineLockingState* utility functions. Suppose that N is the amount of basic regions, M is the amount of field references and method invocations, and P is the amount of collaborating programmers. According to the complexity analysis of the three utility functions presented in Section V-B, the complexity of the *CheckPermission* procedure is $O(N+2(N+P)+NM+N*(N+P))$, which can be further derived as $O(NM+N^2+NP)$. In addition, performance evaluations on this procedure will be presented in Section VI-B.

D. LOCKING STATE UPDATE PROCEDURE

Locking state update plays an essential role in the novel DAL scheme. It is responsible for granting and releasing locks under predefined conditions, by updating values in the locking table.

In recognizing dynamic DG editing operations which may affect programmers' locking scopes, the locking state update procedure needs to be performed with every editing operation executed. Technically, it can be realized with two alternative approaches. The first approach is *operation-based*, where programmers' locking states (i.e., locking scopes) are incrementally updated by extracting and analyzing fine-grained DG changes embedded within each editing operation. However, based on investigations and experiments, we observed that this approach is technically infeasible. The second approach is *state-based*, where each programmer's locking state is derived and updated based on (a) the latest source code content and (b) the latest editing position of the programmer. We have adopted this approach for several reasons. Firstly, the *dependency derivation technique* (as presented in prior work [5]) for deriving source code regions and dependency relationships can be reused for supporting the state-based update, which is efficient enough for execution with every editing operation performed. Secondly, since the same derivation technique will be applied at each collaborating site, the consistent derivation (and thus consistent locking state update) can be technically guaranteed. Thirdly, based on the design of the locking state data structure (i.e., locking table), it is straight-forward and efficient to update a programmer's locking state by simply removing existing recorded locks and inserting newly derived locks with respect to the programmer.

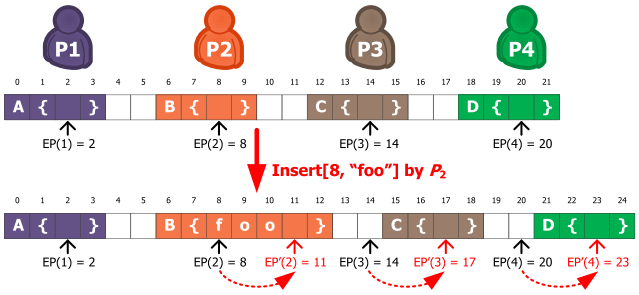

FIGURE 17. Contextualization of editing positions.

One essential element of the above state-based approach is the *editing position* of each programmer, which refers to the programmer's working location in the source code. Let $EP(i)$ be the editing position of the programmer i in the source code document, which is an integer ranging from 0 to $(L-1)$ where L is the length of the source code document. For supporting localized execution of locking state update, each collaborating site maintains a *list of editing positions* $\{EP(i)|i = 1, 2, \ldots, N\}$ for all N programmers (i.e., N distributed sites) in the same collaboration session. In other words, every site has the knowledge of the latest editing positions of all sites in the session, which will be used for locking state update, and for other purposes as well (such as collaboration awareness).

Whenever an editing operation has been executed, a *contextualization* procedure is needed, which updates the editing positions of all programmers based on the evolved document content. Without contextualization, the outdated editing positions will lead to incorrect derivation of working regions and depended regions. For illustrating the necessity of contextualization, a simple example is presented in Fig. 17, where there are four collaborating programmers P_1, P_2, P_3 and P_4 editing methods A, B, C and D respectively, and their initial editing positions are indicated in the upper part. Suppose that programmer P_2 issues an editing operation *Insert*[8, "*foo*"] (which inserts a string "*foo*" at the position 8), and the source code content evolves as presented in the lower part.

It can be intuitively observed that this editing operation not only changes P_2's editing position, but also affects editing positions of P_3 and P_4. Without contextualization, both editing positions of P_3 and P_4 would fall in open areas, which are obviously wrong. In fact, an editing operation, either local or remote, may potentially affect all programmers' editing positions. Thus, the contextualization procedure should be executed following the execution of each local or remote

editing operation. Algorithm 2 implements the procedure *Contextualize(O, i)*, which contextualizes the editing position of programmer i (i.e., site i) with respect to the given editing operation O.

Algorithm 2 Contextualize(O, i)

```
1  if O.type = INSERT
2    if O.position <= EP(i)
3      EP(i) += O.length;
4  else if O.type = DELETE
5    if O.position < EP(i)
6      if O.position + O.length
       <= EP(i)
7        EP(i) -= O.length;
8      else
9        EP(i) := O.position;
```

Under the prior DAL scheme, the locking state update followed a *partial derivation* approach [5]: given an editing operation, only the *operation-actor*'s locking state[5] will be updated. Concretely, for a *local* editing operation, the system only updates the locking state of the *local* site, while for a *remote* editing operation, the system only updates the locking state of the *remote* site that generates this operation. In contrast, under the novel DAL scheme where concurrent and dynamic DG editing operations are possible, the partial derivation approach will no longer be applicable. For example, in the case illustrated by Fig. 14, at programmer P_1's site, after the execution of the remote editing operation issued by P_2, both local and remote programmers' locking states have been affected. It is obvious that an editing operation may affect both the operation-actor and non-operation-actors' locking states under the novel DAL scheme, which requires a *full derivation* approach for locking state update: for each local or remote editing operation executed, the locking states of all collaborating sites in the session should be *fully* derived and updated based on the latest source code content and the contextualized editing positions. Therefore, the entire locking state update scheme can be named as a *contextualization and full derivation* (CFD) scheme, which is formally defined as follows:

Definition 3: Contextualization and Full Derivation (CFD) Locking State Update Scheme

At each collaborating site within the real-time collaboration session, whenever a local or remote editing operation has been executed, the locking states of all collaborating sites in the session are derived and updated as follows:

For each collaborating site i (where $i = 1, 2, \ldots, N$) in the session (which contains N sites):

Step 1. All existing locks (if any) granted to site i are released;

Step 2. $EP(i)$ is contextualized with respect to the executed editing operation;

Step 3. If $EP(i)$ is within the scope of a basic region W, then locks on the working region W and its depended regions are granted to site i; otherwise, no lock is granted to site i.

Algorithm 3 below implements the CFD locking state update procedure *CFD_LSU*(O), which will be invoked following the execution of the editing operation O. In terms of the efficiency, the complexity of this procedure is $O((NP + N + NM + N^2) * P)$, which can be further derived as $O(NP^2 + NMP + N^2P)$, where N is the amount of basic regions, M is the amount of field references and method invocations, and P is the amount of collaborating programmers. In addition, performance evaluations on this procedure will be presented in Section VI-B.

Algorithm 3 CFD_LSU(O)

```
1  for each site i in the session {
2    ReleaseLocks(i);
3    Contextualize(O, i);
4    W := DetectTargetedRegion(EP(i));
5    if W != OA {
6      DRS := DeriveDependedRegionSet
       (W);
7      GrantLocks(i, W, DRS);
8    }
9  }
```

Fig. 18 presents a case that demonstrates the fine-grained effects of the CFD locking state update during a real-time collaboration session. In this case, three programmers P_1, P_2 and P_3 are concurrently working at different locations of the same source code document, and they issued three editing operations during the process: P_1 issues O_1, which creates a dependency relationship $A \rightarrow C$; P_2 issues O_2, which destroys the source code region C; and P_3 issues O_3, which creates a source code region E and a dependency relationship $E \rightarrow A$ simultaneously. At each collaborating site, the CFD locking state update procedure is executed immediately following the execution of each local or remote editing operation. It can be intuitively observed that the novel DAL scheme with the shared-locking approach preserves continuous programming work and correct locking states at any time during the process, and eventually produces consistent locking states over all sites after all editing operations have been propagated and remotely executed.

E. INTEGRATED PROCESSORS OF THE NOVEL DAL SCHEME

In the previous subsections, we presented detailed design of essential building blocks for the novel DAL scheme. As presented in Fig. 19, together with other supporting modules for real-time collaborative programming features, the scheme has been implemented as two integrated processors:

[5]The *operation-actor* of an editing operation refers to the site that generates the operation, whereas a *non-operation-actor* of an editing operation refers to a site that receives and remotely replays the operation.

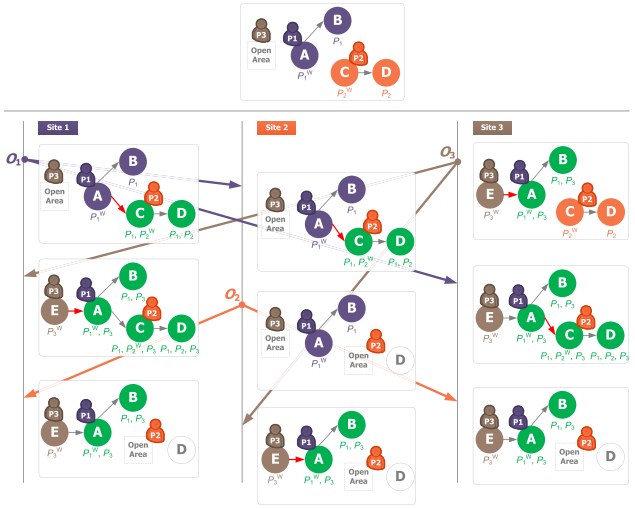

FIGURE 18. Demonstration of the CFD locking state update.

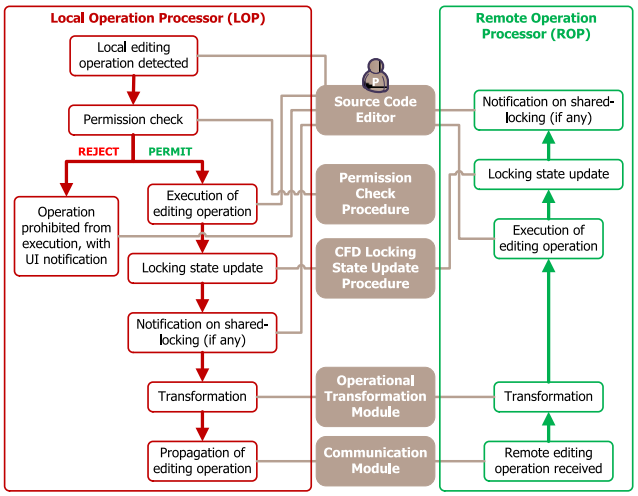

FIGURE 19. Integrated processors of the novel DAL scheme.

(a) the *local operation processor* (LOP), which handles all local editing operations issued by the local programmer; and (b) the *remote operation processor* (ROP), which handles all remote editing operations received. The two processors work in separated and parallel threads at each collaborating site.

Whenever a local editing operation is detected by the system, the permission check procedure will be firstly invoked. If the editing permission is denied, the operation will not take effect on the source code, and the programmer will be notified in the user interface. Otherwise, if the editing permission is granted, the editing operation will immediately take effect on the source code, and then the CFD locking state update procedure will be executed. Eventually, the editing operation will be processed by the operational transformation module for syntactic consistency maintenance issues (as discussed in prior work [4]) and propagated to remote sites.

Whenever a remote editing operation arrives, it will be immediately processed by the operational transformation module and executed on the source code. There is no permission check needed, because the operation has already been permitted by the permission check procedure at its original site, and its permission at a remote site can always be guaranteed under the shared-locking approach. Following its execution, the CFD locking state update procedure will be invoked.

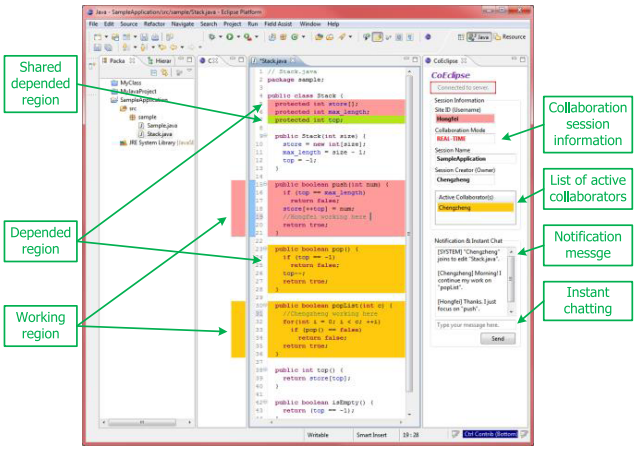

FIGURE 20. A UI snapshot of the research prototype.

VI. PROTOTYPE IMPLEMENTATION AND PERFORMANCE EVALUATIONS

A. PROTOTYPE IMPLEMENTATION

CoEclipse is a research prototype which was initially implemented in prior work [4], [5]. It runs as a plugin of the Eclipse IDE,[6] and provides real-time collaborative programming features (including semantic conflict prevention). Collaborating programmers can use all existing functionalities provided by the Eclipse IDE as usual, and also enjoy extra features contributed by our research.

In this study, the *CoEclipse* prototype has been re-designed for the novel DAL scheme. Concretely, the prototype has been re-implemented with the new locking state data structure (in Section V-A), the utility functions (in Section V-B), the new permission check procedure (in Section V-C), the CFD locking state update procedure (in Section V-D), and the integrated LOP and ROP of the novel DAL scheme (in Section V-E). In addition, several collaboration awareness features, such as the shared-locking notification and locking state awareness, have been integrated in the user interface of the IDE. Fig. 20 presents a UI snapshot when two programmers are concurrently editing the same Java source code document (the *Stack* implementation) using the *CoEclipse* prototype. At this moment, the local programmer is editing the *push* method while the remote programmer is editing the *popList* method, and their locking states (i.e., the granted locks on working regions and depended regions) can be intuitively observed from the colored highlights on various source code regions.

[6]http://www.eclipse.org/ide

TABLE 1. Performance evaluations on utility functions related to source code content.

Unit: millisecond

Utility Function	N = 50 M = 2,000	N = 100 M = 4,000	N = 200 M = 8,000	N = 400 M = 16,000	N = 800 M = 32,000
DetectTargetedRegion	0.000023	0.000048	0.000097	0.000211	0.000502
DeriveDependedRegionSet	0.0445	0.0956	0.2512	0.7570	1.9072

N: amount of basic regions (i.e., fields and methods)
M: amount of field references and method invocations

The prototype implementation has confirmed the technical feasibility of the novel DAL scheme with the shared-locking approach. Based on preliminary user evaluations, there is no blocking perceived by the programmers when issuing editing operations in the source code editor. Each editing operation can immediately take effect (i.e., permission granted) or be rejected (i.e., permission denied). This indicates that the local responsiveness of the prototype system is as good as the single-user Eclipse IDE without the *CoEclipse* plugin. Moreover, the execution of remote editing operations and the update of locking states can also be performed efficiently without perceived delay. In general, the DAL features have not incurred any negative performance issue from the end-users' perspective.

B. PERFORMANCE EVALUATIONS

In addition to preliminary user evaluations, we have further designed and conducted a comprehensive set of performance evaluations on utility functions and essential procedures of the novel DAL scheme, which are critical to the overall system performance. Since the DAL features have been implemented on top of an existing system (i.e., the Eclipse IDE), we focus on the additional cost incurred by the DAL mechanisms.

Based on the complexity analysis of the utility functions and essential procedures, the amount of basic regions and dependency relationships in the source code document is a key factor in determining the DAL system performance. For systematically evaluating how the system performance changes with the increase of the amount of basic regions and dependency relationships, as well as for measuring the system performance in extreme (unrealistic) cases, we have implemented a utility tool that can generate Java source code documents based on customizable parameters, such as the amount of basic regions (i.e., fields and methods) and the amount of dependency relationships (i.e., field references and method invocations) in the class. With multiple groups of source code documents as the input data, a comprehensive set of performance experiments have been conducted in a simulation-based approach. The experimental platform is a medium-level desktop PC with a processor of Intel Core i7-3770 @ 3.40 GHz and 4 GB of RAM.

Firstly, we have evaluated the performance of utility functions *DetectTargetedRegion* and *DeriveDependedRegionSet* (in Section V-B), which are related to the source code content. As the performance of the two functions depends on the amount of basic regions (denoted as N) and the amount of field references and method invocations (denoted as M) in the source code, 5 groups of experiments have been conducted with source code documents having different N and M values, and the execution times of the two functions have been measured. Table 1 presents the experimental results, which have confirmed the good performance of both functions. As presented, the average execution times of the functions grow steadily with the increase of N and M values. Even in the extreme case where the source code contains 800 basic regions with 32,000 field references and method invocations (which is unrealistic), the *DetectTargetedRegion* function costs only $5.02*10^{-4}$ ms, and the *DeriveDependedRegionSet* function costs only 1.9072ms.

Secondly, we have evaluated the performance of the three utility functions related to the locking state data structure (i.e., locking table), namely *ExamineLockingState*, *GrantLocks* and *ReleaseLocks* (in Section V-B). In addition to the amount of basic regions (denoted as N) and the amount of field references and method invocations (denoted as M), their performance also depends on the amount of collaborating programmers (denoted as P) in the collaboration session. Accordingly, 20 groups of experiments have been conducted with source code documents having different N and M values and real-time collaboration sessions having different P values, while the execution times of the three functions have been measured. Table 2 presents the experimental results, which have confirmed the good performance of the three functions. As presented, the average execution time of each function grows steadily with the increase of N, M and P values. Even in the worst case where the collaboration session contains 16 collaborating programmers and the source code contains 800 basic regions with 32,000 field references and method invocations (which is unrealistic), the *ExamineLockingState* function costs $8.9 * 10^{-6}$ ms, while the *GrantLocks* and *ReleaseLocks* functions cost $9.916*10^{-3}$ ms and $6.650*10^{-3}$ ms respectively, which are very efficient.

Thirdly, we have evaluated the overall performance of the two essential DAL procedures, namely *CheckPermission* and *CFD_LSU*. The execution times of the two procedures have been measured by 20 groups of experiments with different source code documents and real-time collaboration sessions. Table 3 presents the experimental results, which have confirmed the good performance of the two procedures.

TABLE 2. Performance evaluations on utility functions related to locking state data structure.

Unit: 10^{-3} *millisecond*

P	Utility Function	N = 50 M = 2,000	N = 100 M = 4,000	N = 200 M = 8,000	N = 400 M = 16,000	N = 800 M = 32,000
2	*ExamineLockingState*	0.00316	0.00337	0.00356	0.00372	0.00439
	GrantLocks	0.186	0.422	1.156	3.187	7.923
	ReleaseLocks	0.146	0.308	0.688	1.502	2.919
4	*ExamineLockingState*	0.00357	0.00376	0.00405	0.00435	0.00514
	GrantLocks	0.188	0.432	1.212	3.231	8.407
	ReleaseLocks	0.166	0.352	0.842	1.859	3.469
8	*ExamineLockingState*	0.00424	0.00464	0.00483	0.00528	0.00634
	GrantLocks	0.190	0.436	1.280	3.298	9.152
	ReleaseLocks	0.216	0.444	1.138	2.460	4.405
16	*ExamineLockingState*	0.00562	0.00610	0.00663	0.00748	0.00890
	GrantLocks	0.198	0.453	1.402	3.631	9.916
	ReleaseLocks	0.303	0.623	1.714	3.853	6.650

N: amount of basic regions (i.e., fields and methods)
M: amount of field references and method invocations
P: amount of collaborating programmers in the session

TABLE 3. Performance evaluations on essential procedures of the novel DAL scheme.

Unit: millisecond

P	Procedure	N = 50 M = 2,000	N = 100 M = 4,000	N = 200 M = 8,000	N = 400 M = 16,000	N = 800 M = 32,000
2	*CheckPermission*	0.000029	0.000055	0.000109	0.000223	0.000518
	CFD_LSU	0.0864	0.1909	0.5249	1.5122	3.7331
4	*CheckPermission*	0.000030	0.000057	0.000114	0.000228	0.000529
	CFD_LSU	0.1762	0.3732	1.0140	3.0132	7.4968
8	*CheckPermission*	0.000032	0.000058	0.000115	0.000232	0.000538
	CFD_LSU	0.3503	0.7437	2.0438	6.2269	15.4252
16	*CheckPermission*	0.000033	0.000060	0.000116	0.000239	0.000561
	CFD_LSU	0.6967	1.4959	4.1162	12.4635	30.6650

N: amount of basic regions (i.e., fields and methods)
M: amount of field references and method invocations
P: amount of collaborating programmers in the session

As presented, the average execution times of both procedures grow steadily with the increase of N, M and P values. Even in the extreme case (which is unrealistic), the *CheckPermission* procedure costs only $5.61 * 10^{-4}$ ms, and the *CFD_LSU* procedure costs only 30.665 ms.

Based on the experimental results, it can be concluded that the DAL features, built on top of the existing single-user programming environment, have incurred very low cost in supporting semantic conflict prevention. The efficient execution of DAL mechanisms contributes to the high local responsiveness of the client system and the real-time execution of remote operations, which further contributes to the overall user experience of real-time collaborative programming environments.

VII. COMPARISON WITH RELATED TECHNIQUES AND STUDIES

Firstly, from the general perspective of locking, we compare the DAL scheme with traditional locking schemes in database and distributed systems. One common point is that locking is applied for achieving mutual exclusion in all scenarios: under the DAL scheme, locking helps to prevent potential semantic conflicts by prohibiting concurrent editing on the same source code region and multiple source code regions with dependency relationships; and similarly, in database and distributed systems, locking ensures data integrity by prohibiting concurrent updates on shared objects and resources [1]. However, compared to those traditional locking schemes, the DAL scheme can be distinguished in the following aspects:

1. Locking under the DAL scheme is used for coordinating behaviors among human users (i.e., collaborating programmers), whereas locking in database and distributed systems is commonly used to coordinate pre-programmed processes. As mentioned in Section IV-D, this is one important reason why the shared-locking approach could be appropriate for the DAL scheme.
2. Under the DAL scheme, locks are requested and granted/denied in a non-blocking manner, because all locking mechanisms (e.g., permission check, locking state update) are fully distributed and localized, which ensures the client responsiveness, and more importantly, eliminates the possibility of deadlock. In contrast, in database and distributed systems, a pre-programmed process requesting a lock is commonly blocked until the decision (i.e., granted/denied) is received, and there may exist potential risk of deadlock under those traditional locking schemes. As mentioned in Section IV-D, it is the shared-locking approach that plays an important role in supporting fully distributed, localized and responsive locking.
3. In the context of real-time collaborative programming, the *shared-read-permission* on all source code regions is implicitly granted to all programmers at any time by nature. The DAL scheme only controls the *write-permission* on source code regions. To edit a source code region, a programmer must obtain the *write-permission* (automatically granted by DAL) on it. In the face of concurrent or dynamic DG editing operations, whenever a shared-lock is granted on an overlapping working region, it is essentially a *shared-write-permission* on the region for the involved programmers, while other programmers still have the *shared-read-permission* on the region implicitly. In contrast, in database and distributed systems, the traditional locking schemes control both of the *write-permission* and *read-permission*, and a shared-lock commonly refers to a *shared-read-permission* on a resource/object, while there is usually no *shared-write-permission* allowed.

Secondly, from the perspective of consistency maintenance in real-time collaborative editing, we compare the DAL scheme with other consistency maintenance techniques. As reviewed in Section I, to achieve local responsiveness, the replicated architecture has been commonly adopted [22], and consequently, there arises the need for syntactic consistency maintenance of the replicated document copies. The CSCW community has contributed a variety of approaches and techniques for supporting syntactic consistency maintenance in real-time collaborative editing, such as the *operational transformation* (OT) techniques [2], [16], [19], [21], [24], [28], [29] and the *address space transformation* (AST) techniques [9], [10], [12], [27], [30]. The DAL scheme in this study is significantly distinguished from the abovementioned techniques, because it is purely responsible for semantic consistency maintenance (or namely semantic conflict prevention), which is another category of consistency issues (as reviewed in Section I and discussed in prior work [5], [21], [23]). In this study, the OT technique has been adopted for achieving syntactic consistency maintenance in real-time collaborative programming, while the semantic consistency maintenance is supported by the DAL scheme.

Thirdly, from the specific perspective of semantic consistency maintenance, we compare the DAL scheme with other locking schemes for supporting similar purposes. One previous study [23] proposed a locking scheme for supporting semantic consistency maintenance in real-time collaborative plain-text editing systems, where collaborating users are allowed to request locks on selected textual ranges. A follow-up study [20] further contributed two protocols for resolving locking operation conflicts in the system. The DAL scheme in this study is significantly different from the abovementioned studies in the following major aspects:

1. Under the DAL scheme, all locks are automatically derived, granted and released, without any manual effort from the programmers; whereas in the previous studies, locks were always manually requested and released, requiring additional coordination effort from the users.
2. In the previous studies, each lock was placed on a sequence of consecutive characters in the shared plain-text document. When several sequences need to be locked at once, a user has to either request multiple locks on several sequences, or request one lock on a potentially large scope covering those sequences (but also covering many sequences which are not intended to lock, leading to reduced concurrency). In contrast, the DAL scheme in this study provides intelligent and fine-grained locking, where locks are always granted and released as a group, covering the latest working region and depended regions only, which supports a reasonable balance between conflict prevention and concurrent work. More importantly, the derivation of the working region and depended regions are completely automatic, without any manual effort needed.

VIII. CONCLUSIONS AND FUTURE WORK

Dependency-based automatic locking (DAL) is a novel approach for supporting semantic conflict prevention in real-time collaborative programming, which was proposed in prior work. However, the prior DAL scheme was devised under two assumptions, which are not realistic in practice. Together with other restrictions, they become serious problems in applying the DAL techniques in real-world programming scenarios. To address the issues, we have proposed a novel DAL scheme with a shared-locking approach, which ensures the responsiveness, effectiveness and consistency of semantic conflict prevention in unconstrained real-time collaborative programming, where programmers are allowed to perform concurrent editing operations with overlapping locking scopes, and to perform editing operations that dynamically change the source code structure.

Based on detailed analysis of the problems under the prior DAL scheme, we have proposed a shared-locking approach which consists of three types of shared-locking allowed under well-defined circumstances, with detailed design rationales in four aspects. The shared-locking approach (a) ensures high local responsiveness and minimum overhead while maintaining consistent locking states over all sites, (b) preserves continuous work for all programmers under all circumstances, (c) ensures the correctness and effectiveness of semantic conflict prevention, and (d) complies with the nature of real-time collaborative programming environments. We have presented major technical issues and solutions in realizing the novel DAL scheme, including the locking state data structure, the utility functions, the permission check procedure, the locking state update procedure, and the integrated processors for handling local and remote editing operations. We have implemented the scheme in a research prototype, and conducted a comprehensive set of performance evaluations on the utility functions and essential procedures, which have confirmed the good performance.

We have been continuously working in the field of semantic conflict prevention for real-time collaborative programming environments, and our future work includes: (a) semantic conflict prevention across multiple documents; (b) flexible locking scope determination with fine-grained strength measurement of dependency relationships for better balancing conflict prevention and concurrent work; and (c) further improvement of the research prototype for more evaluations.

REFERENCES

[1] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981.
[2] W. Cai, F. He, and X. Lv, "Multi-core accelerated operational transformation for collaborative editing," in *Proc. Int. Conf. Collaborative Comput., Netw., Appl. Worksharing*, 2015, pp. 121–128.
[3] Y. Chen, S. W. Lee, Y. Xie, Y. Yang, W. S. Lasecki, and S. Oney, "Codeon: On-demand software development assistance," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2017, pp. 6220–6231.
[4] H. Fan and C. Sun, "Achieving integrated consistency maintenance and awareness in real-time collaborative programming environments: The CoEclipse approach," in *Proc. IEEE 16th Int. Conf. Comput. Supported Cooperat. Work Design (CSCWD)*, May 2012, pp. 94–101.
[5] H. Fan and C. Sun, "Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming," in *Proc. 27th Annu. ACM Symp. Appl. Comput.*, 2012, pp. 737–742.
[6] H. Fan, C. Sun, and H. Shen, "ATCoPE: Any-time collaborative programming environment for seamless integration of real-time and non-real-time teamwork in software development," in *Proc. 17th ACM Int. Conf. Supporting Group Work*, 2012, pp. 107–116.
[7] H. Fan, H. Zhu, Q. Liu, Y. Shi, and C. Sun, "Shared-locking for semantic conflict prevention in real-time collaborative programming," in *Proc. IEEE 21st Int. Conf. Comput. Supported Cooperat. Work Design (CSCWD)*, 2017, pp. 174–179.
[8] M. S. Feldman, "CodeSync: A collaborative coding environment for novice Web developers," M.S. thesis, Wellesley College, Wellesley, MA, USA, 2014.
[9] L. Gao, F. Yu, Q. Chen, and N. Xiong, "Consistency maintenance of do and undo/redo operations in real-time collaborative bitmap editing systems," *Cluster Comput.*, vol. 19, no. 1, pp. 255–267, 2016.
[10] L. Gao, F. Yu, L. Gao, N. Xiong, and G. Yang, "Consistency maintenance of compound operations in real-time collaborative environments," *Comput. Elect. Eng.*, vol. 50, pp. 217–235, Feb. 2016.
[11] M. Goldman, G. Little, and R. C. Miller, "Real-time collaborative coding in a Web IDE," in *Proc. 24th Annu. ACM Symp. User Interface Softw. Technol.*, 2011, pp. 155–164.
[12] N. Gu, J. Yang, and Q. Zhang, "Consistency maintenance based on the mark & retrace technique in groupware systems," in *Proc. Int. ACM SIGGROUP Conf. Supporting Group Work*, 2005, pp. 264–273.
[13] P. J. Guo, J. White, and R. Zanelatto, "Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning," in *Proc. IEEE Symp. Vis. Lang. Hum.-Centric Comput. (VL/HCC)*, Oct. 2015, pp. 79–87.
[14] A. Kurniawan, C. Soesanto, and J. E. C. Wijaya, "CodeR: Real-time code editor application for collaborative programming," *Procedia Comput. Sci.*, vol. 59, pp. 510–519, Jan. 2015.
[15] J. Lautamäki, A. Nieminen, J. Koskinen, T. Aho, T. Mikkonen, and M. Englund, "CoRED: Browser-based collaborative real-time editor for java Web applications," in *Proc. ACM Conf. Comput. Supported Cooperat. Work*, 2012, pp. 1307–1316.
[16] A. Ng and C. Sun, "Operational transformation for real-time synchronization of shared workspace in cloud storage," in *Proc. 19th Int. Conf. Supporting Group Work*, 2016, pp. 61–70.
[17] V. Nguyen, H. H. Dang, N.-K. Do, and D.-T. Tran, "Enhancing team collaboration through integrating social interactions in a Web-based development environment," *Comput. Appl. Eng. Edu.*, vol. 24, no. 4, pp. 529–545, 2016.
[18] M. R. J. Rantala, "Real-time collaborative coding—Technical and group work challenges," M.S. thesis, Faculty Bus. Built Environ., Tampere Univ. Technol., Tampere, Finland, 2015.
[19] B. Shao, D. Li, T. Lu, and N. Gu, "An operational transformation based synchronization protocol for Web 2.0 applications," in *Proc. ACM Conf. Comput. Supported Cooperat. Work*, 2011, pp. 563–572.
[20] C. Sun, "Optional and responsive fine-grain locking in Internet-based collaborative systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 9, pp. 994–1008, Sep. 2002.
[21] C. Sun. (2015). *OTFAQ: Operational Transformation Frequently Asked Questions and Answers*. [Online]. Available: http://www.ntu.edu.sg/home/czsun/projects/otfaq
[22] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, no. 1, pp. 63–108, 1998.
[23] C. Sun and R. Sosič, "Optimal locking integrated with operational transformation in distributed real-time group editors," in *Proc. 18th Annu. ACM Symp. Principles Distrib. Comput.*, 1999, pp. 43–52.
[24] C. Sun, Y. Xu, and A. Ng, "Exhaustive search and resolution of puzzles in OT systems supporting string-wise operations," in *Proc. ACM Conf. Comput. Supported Cooperat. Work Social Comput.*, 2017, pp. 2504–2517.
[25] Y. Wang, P. Wagstrom, E. Duesterwald, and D. Redmiles, "New opportunities for extracting insights from cloud based IDEs," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 408–411.
[26] J. Warner and P. J. Guo, "CodePilot: Scaffolding end-to-end collaborative software development for novice programmers," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2017, pp. 1136–1141.
[27] H. Xia, T. Lu, B. Shao, G. Li, X. Ding, and N. Gu, "A partial replication approach for anywhere anytime mobile commenting," in *Proc. 17th ACM Conf. Comput. Supported Cooperat. Work Social Comput.*, 2014, pp. 530–541.
[28] Y. Xu and C. Sun, "Conditions and patterns for achieving convergence in OT-based co-editors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 695–709, Mar. 2016.
[29] Y. Xu, C. Sun, and M. Li, "Achieving convergence in operational transformation: Conditions, mechanisms and systems," in *Proc. 17th ACM Conf. Comput. Supported Cooperat. Work Social Comput.*, 2014, pp. 505–518.
[30] D. Yang, T. Lu, H. Xia, B. Shao, and N. Gu, "Making itinerary planning collaborative: An AST-based approach," in *Proc. IEEE 20th Int. Conf. Comput. Supported Cooperat. Work Design (CSCWD)*, May 2016, pp. 257–262.

HONGFEI FAN received the B.E. degree in software engineering from Tongji University, China, in 2007, and the Ph.D. degree in computer science from Nanyang Technological University, Singapore, in 2013.

Since 2014, he has been an Assistant Professor with the School of Software Engineering, Tongji University, China. His research interests include computer-supported cooperative work and software engineering.

HONGMING ZHU received the Ph.D. degree in computer science from the University of Bolton, U.K, in 2014.

He is currently a Lecturer with the School of Software Engineering, Tongji University, China. His research interests include distributed computing and software engineering.

QIN LIU received the B.S. degree in industrial automation from the Dalian University of Technology, China, in 1998, the M.S. degree in software engineering from Southampton Solent University, U.K., in 2001, and the Ph.D. degree in software engineering from Northumbria University, U.K., in 2006.

She is currently a Professor with the School of Software Engineering, Tongji University, China. From 2008 to 2017, she served as the Executive Dean of the School. Her research interests include software engineering and software testing.

YANG SHI received the B.S. degree in electronic engineering from the Hefei University of Technology, China, in 1999, the M.S. degree in pattern recognition and intelligence systems from the Kunming University of Science and Technology, China, in 2002, and the Ph.D. degree in pattern recognition and intelligent systems from Tongji University, China, in 2005.

From 2005 to 2011, he was with Pudong CS&S Co., Ltd., Shanghai, China. Since 2012, he has been an Associate Professor with the School of Software Engineering, Tongji University, China. His research interests include software engineering and information security.

CHENGZHENG SUN received the Ph.D. degree in computer engineering from the National University of Defense Technology, China, in 1987, and the second Ph.D. degree in computer science from the University of Amsterdam, Netherlands, in 1992.

He is currently a Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His current research lies in the intersections of computer-supported cooperative work and distributed systems.