Hybrid Semantic Conflict Prevention in Real-Time Collaborative Programming

Wenhua Xu[1], Yiteng Zhang[1], Brian Chiu[1], Dong Chen[2], Jinfeng Jiang[1], Bowen Du[3(✉)], and Hongfei Fan[1(✉)]

[1] School of Software Engineering, Tongji University, Shanghai, China
{2031534,1852137,1850250,1751047,fanhongfei}@tongji.edu.cn
[2] SAP Labs China, Shanghai, China
dong.chen03@sap.com
[3] Department of Computer Science, University of Warwick, Coventry, UK
B.Du@warwick.ac.uk

Abstract. Real-time collaborative programming allows a group of programmers to edit the same source code at the same time. To support semantic conflict prevention in real-time collaboration, a dependency-based automatic locking (DAL) approach was proposed in prior work. The DAL mechanism automatically detects programming elements with dependency relationships, and prohibits concurrent editing on the inter-dependent source code regions by locking. However, the prior DAL scheme is too restrictive, which leads to an unnecessarily large locking scope and seriously impacts the concurrent work. To address this issue, we propose a novel hybrid semantic conflict prevention (HSCP) scheme, to achieve a better balance between conflict prevention and concurrent work. The scheme enforces locking on the working and strongly-depended regions, while applies awareness highlight on weakly-depended regions. The depth of locking scope can be customized by each programmer in a fine-grained manner. A three-level awareness mechanism has been designed for programmers to intuitively distinguish working, strongly-depended and weakly-depended regions. In supporting the scheme, we have devised techniques and solutions, and implemented a prototype that supports programmers to enjoy hybrid semantic conflict prevention in real-time collaborative programming over Eclipse and IntelliJ IDEA platforms. Experimental evaluations have confirmed the satisfactory performance of the scheme in complex real-world scenarios.

Keywords: Real-time collaborative programming · Hybrid Semantic Conflict Prevention (HSCP) · Dependency-Based Automatic Locking (DAL) · Customizable locking scope determination · Collaboration awareness mechanism

1 Introduction

Software development requires effective collaboration by multiple programmers. In general, there are two approaches for collaborative programming, namely non-

H. Gao and X. Wang (Eds.): CollaborateCom 2021, LNICST 407, pp. 104–123, 2021.
https://doi.org/10.1007/978-3-030-92638-0_7

real-time collaborative programming and real-time collaborative programming [5]. Non-real-time collaborative programming has been widely applied in the industry as a traditional and mature collaboration approach, which is commonly supported by version control systems like Git. In contrast, real-time collaborative programming is an emerging collaborative programming approach that has attracted increasing attention and interests in the recent years [1–3,8–13,16,17]. In a real-time collaborative programming session, each programmer's editing operations will be immediately propagated to all other collaborators' sites [15]. Real-time collaborative programming allows each collaborator to clearly know the work progress of others, reduces the cost of communication, and increases the efficiency of work. In addition, there is no need to manually merge concurrent work, and conflicts due to communication problems could be reduced. As an emerging approach, real-time collaborative programming is an effective supplement to the traditional non-real-time collaborative programming, and both approaches collectively meet diverse collaboration needs at different stages of software development processes [6].

The generic real-time collaborative editing technique is an essential part of implementing collaborative programming systems. In order to allow collaborators to edit freely in a session without being affected by network factors, real-time collaborative editing systems have commonly been designed with a *replicated architecture*, where each collaborating site maintains a distributed copy of the shared document. Under such architecture, each user's local editing operation will be immediately executed at the local site and sent to other collaborators. Consequently, there arises a classic problem called *syntactic consistency* [14], which is concerned with maintaining the consistency of all distributed copies of the document after all editing operations have been propagated and executed remotely. In order to solve this problem, a sophisticated *operational transformation* (OT) technique has been proposed and widely adopted [14,15]. By transforming the remote operations under certain conditions, the syntactic consistency of the replicated document can be guaranteed at all collaborators' sites.

In addition, there is another higher-level consistency problem, named *semantic consistency*. Syntactic consistency considers the consistency of the replicated code, whereas semantic consistency emphasizes the consistency of coding logic [4,7,14]. For supporting semantic consistency maintenance (also regarded as semantic conflict prevention), prior work proposed a *dependency-based automatic locking* (DAL) scheme [7], which prevents concurrent editing work on the same and inter-dependent source code regions. However, the existing DAL scheme is too restrictive: whenever a user is editing a basic region, all depended regions will be locked by the user until he/she finishes, and concurrent work is seriously impacted by the extremely large locking scope.

To better balance semantic conflict prevention and concurrent programming work, we contribute a novel *Hybrid Semantic Conflict Prevention* (HSCP) approach in this study. Under the HSCP scheme, the depended regions of each programmer are divided into strongly-depended and weakly-depended regions. Locking is enforced in working regions and strongly-depended regions, whereas

awareness highlight is applied in weakly-depended regions. In addition, the depth of the scope of strongly-depended regions (i.e. locking scope) can be customized by each individual programmer in a fine-grained manner. The HSCP scheme also employs a three-level awareness mechanism, which assists collaborating programmers to intuitively distinguish working regions, strongly-depended regions, and weakly-depended regions, helping programmers to assess the risk of semantic conflicts. All approaches, mechanisms and techniques have been successfully implemented in prototype systems, while user evaluations and experimental evaluations have confirmed the feasibility, effectiveness, efficiency and usability of the system.

The rest of this paper is organized as follows. Firstly, we briefly review prior work on semantic conflict prevention with the DAL scheme in Sect. 2. Secondly, we provide detailed analysis on the constraints of the previous DAL scheme in Sect. 3. Thirdly, we propose the HSCP scheme in Sect. 4. Consequently, we present major technical issues and solutions for implementing the HSCP scheme in Sect. 5. We demonstrate the prototype implementation, experimental evaluations, and preliminary user evaluations in Sect. 6. Finally, we summarize this study in Sect. 7.

2 Related Work: Review of Prior Work on Semantic Conflict Prevention with Dependency-Based Automatic Locking (DAL)

Based on investigations, when multiple programmers are working in a real-time collaborative programming session, semantic conflicts may occur. Different from other collaborative work such as text editing, programming work requires strict consistency and continuity in terms of coding logic. Due to these characteristics of programming work, it is easy for programmers to unintentionally modify the logic of other programmer's code in a real-time collaborative programming session. When collaborative programmers are working in the same source code region or multiple source code regions with dependency relationships, semantic conflicts may occur [4,7]. For example, when two programmers P_1 and P_2 are concurrently editing the same Java source code document (*Queue* implementation) as shown in Fig. 1, semantic conflicts may occur if P_1 and P_2 are concurrently editing method *offer*. In addition, semantic conflicts may also occur when two programmers are editing different regions with a dependency relationship (e.g., P_1 is editing the method *isEmpty*, while P_2 is editing the method *poll* which invokes *isEmpty*.).

For the convenience of discussion, we firstly introduce several terms as follows [4,7]:

1. In object-oriented programming languages (such as Java), a class can be divided into *open areas* and *basic regions*. A basic region is a self-contained logical unit, which can refer to a *field* or a *method*; while all other spaces can be regarded as open areas.

2. In a source code document, if basic region A depends on region 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. If $A \not\rightarrow B$ and $B \not\rightarrow A$, A and B are independent. The dependency relationship is transitive, when $A \rightarrow B$, $B \rightarrow C$, then $A \rightarrow C$. For example, if A is a method, the field B is referenced in A, $A \rightarrow B$, the method C invoking A, $C \rightarrow A$, and then $C \rightarrow B$.
3. Given a source code document, we can obtain a *dependency graph* (DG) by analyzing the dependency relationship among all basic regions. In the DG, a node represents a basic region, and an edge from node A to node B represents a dependency relationship from node A to node B, denoted as $A \rightarrow B$.

The source code *Queue.class* can be parsed into a DG with 7 open areas and 10 basic regions as shown in Fig. 1. It is worth further mentioning that any piece of source code that cannot form a valid basic region will be treated as an open area, including those intended but uncompleted regions. For example, when a Java method is being created but not completed yet (e.g. a brace is missing), the code segment will be regarded as an open area until it is syntactically completed.

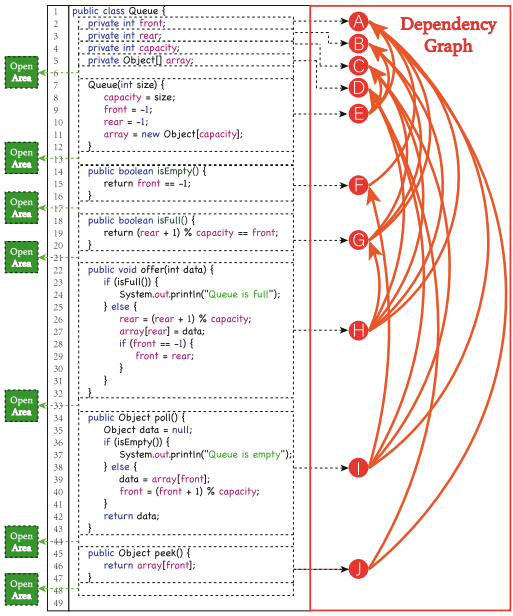

Fig. 1. Basic regions, dependency relationships and dependency graph (DG)

In prior work, to solve the problem of semantic conflict, a *Dependency-based Automatic Locking* (DAL) approach was proposed [4,7]. Whenever a programmer attempts to edit a region, the DAL scheme automatically detects the working region, derives the depended regions, requests and grants locks on them

for this programmer, and finally releases locks when the work is completed. The whole mechanism works without any manual effort from the programmers. After each local or remote editing operation is executed, a *Contextualization and Full Derivation Locking State Update* (CFD_LSU) procedure will be executed [7], which ensures correct locking on the latest working region and depended regions for each programmer at any time, based on the programmers' latest editing positions and the latest DG structure. Under the DAL scheme, the *CheckPermission* procedure can prevent programmers from accidentally working on the regions which may cause semantic conflicts [7]. When a programmer selects a basic region to start working, if the basic region is locked by other programmers, or there is another user's working region within the depended regions, the editing operation will be rejected along with a UI notification as shown in Fig. 2.

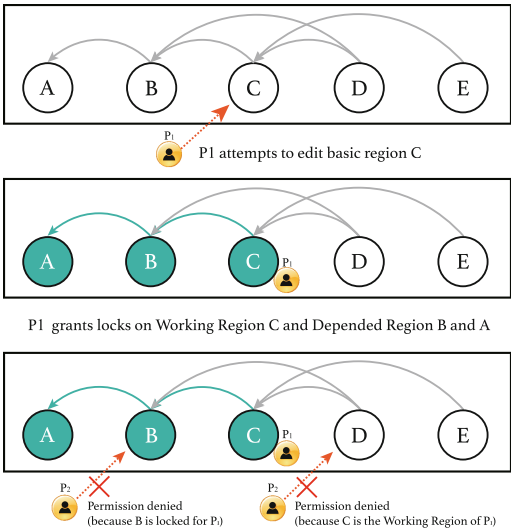

Fig. 2. A simple example of the DAL permission check mechanism

3 Major Constraints of the Prior DAL Scheme

The prior DAL scheme focused on the prevention of semantic conflicts without considering user experience in a real-world software development scenario. Whenever a programmer starts to work on a non-locked basic region, the working region and all depended regions will be locked. Due to the transitivity of dependency relationships, it is easy to produce extremely large locking scope. Consequently, any other programmer's editing operation on the locked regions will be rejected, which greatly affects the concurrency of collaborative work among programmers, especially for small-scale software development scenarios (e.g. multiple programmers working in the same class or package).

On the one hand, an extremely large locking scope may not be necessary for semantic conflict prevention. It is intuitive that semantic conflicts are more likely to occur among regions with stronger dependency relationships, which form a relatively small locking scope. Given a working region, it is possible that most depended regions are indirectly depended by the working region, which are far away from the working region along the dependency paths. Locking these weakly-depended regions does not make much sense, but greatly reduces the concurrency of programming work. On the other hand, shrinking the locking scope will inevitably weaken the capability of semantic conflict prevention. In certain cases, semantic conflicts may occur even if programmers work in regions with weak dependency relationships. It is not worth locking basic regions with weak dependency relationships; however, if no measure is taken, semantic conflicts may still occur.

Based on these observations, it is preferable if the locking scope could be reasonably reduced for better balancing concurrent work and conflict prevention, while at the same time, a weaker measure of semantic conflict prevention (weaker than locking) could be applied on those weakly-depended regions. It is also beneficial if the DAL scheme could support a general mechanism for more fine-grained locking scope determination where the depth of locking scope can be customized by each collaborating programmer.

4 HSCP: Hybrid Semantic Conflict Prevention

To solve the various constraints in the prior DAL scheme and improve the usability of semantic conflict prevention, we propose a *Hybrid Semantic Conflict Prevention* (HSCP) scheme, which employs a hybrid approach combining locking and awareness.

For the convenience of discussion, we introduce a term named dependency depth and a concept of three type regions as follows:

– In the DG, the *dependency depth* from node A to node B refers to the minimum number of edges from node A to node B in the DG. For example, $A \rightarrow B \rightarrow C$ and $A \rightarrow E \rightarrow F \rightarrow C$, and then the dependency depth from $A \rightarrow C$ is 2. If $A \not\rightarrow D$, then the dependency depth of A to D is regarded as infinite. A simple example of dependency depth is presented in Fig. 3.

Based on investigations, it is observed that the risk of semantic conflicts is closely related to the dependency depth. Given a working region, semantic conflicts are more likely to occur in depended regions with smaller dependency depths from that working region, while semantic conflicts are less likely to occur in regions with larger dependency depths. It is necessary to determine a limited locking scope by locking regions within a reasonable dependency depth. Due to the complexity of software development, it is impossible to specify a general locking depth for all occasions, and it is preferable that the DAL scheme provides a generic working mechanism but leaving the setting of dependency depth up to the collaborating programmers. For example, if collaborating programmers

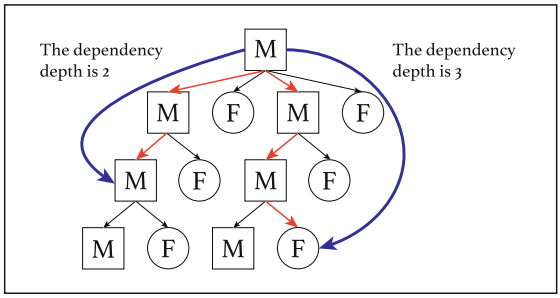

Fig. 3. A simple example of dependency depth

judge that there is less risk of semantic conflicts in the current session and more concurrent work is needed to speed up the work progress, the programmers may appropriately choose a smaller locking scope by setting a smaller dependency depth.

Therefore, we propose a customizable locking scope determination mechanism, where each programmer can customize his/her depth of the locking scope freely by setting the locking scope policy. Within the policy setting, there are two important arguments, namely F-depth and M-depth, which can precisely determine the locking scope of depended regions. All arguments of the policy setting are presented in Table 1 below.

Table 1. Arguments of customizable locking scope policy setting

Argument	Description
Locking Depth Switch	If it is turned on, the locking scope is determined according to F-depth and M-depth;
	If it is turned off, all depended regions will be locked (similar to the prior DAL scheme).
F-depth	Locking all field regions with dependency depth $\leq$ F-depth
M-depth	Locking all method regions with dependency depth $\leq$ M-depth

In addition, we define three types of regions which have different risk of semantic conflicts. When a programmer is editing a basic region, the set of depended regions of this region can be divided into the working regions, depended regions within the locking scope (denoted as *strongly-depended regions*), and the depended regions outside the locking scope (denoted as *weakly-depended regions*).

In real-world scenarios, each collaborating programmer can customize F-depth and M-depth in the user interface to dynamically adjust the locking scope. The change of locking scope can be immediately propagated to all collaborators' sites. It is worth pointing out that each collaborating programmer has his/her own locking scope policy and can only be decided by himself/herself. This implies that in a collaborative programming session, there may be different locking scope policies corresponding to different users at the same time. Although the customizable locking scope mechanism is much more complicated than a global

locking scope policy (i.e. all users having the same locking scope policy), it is more fine-grained and has the following two advantages.

1. A global policy setting may not comply with the nature of collaborative work. If one user modifies the global policy setting, the locking scope of all users will be changed, which may interrupt the work of others and cause the originally-locked depended regions to be unlocked and modified by others.
2. A global policy setting is not flexible enough. The programming preferences of programmers may be diverse and variant. Some programmers may prefer a larger locking scope (with less risk of semantic conflicts), whereas other programmers may choose a small locking scope with more concurrency of work. A global policy setting obviously cannot satisfy all programmers.

For the above reasons, the customizable locking scope determination is very flexible and can well enhance the user experience. To intuitively explain the customizable locking scope determination, we present a simple example as shown in Fig. 4. In the figure, two programmers are concurrently working in the same document and have different locking scope policy settings. Under the prior DAL scheme, multiple programmers were certainly not allowed to work at the same time (because each user needs to lock all depended regions).

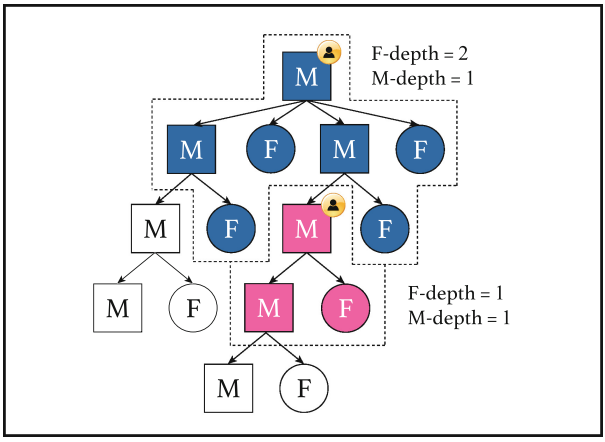

Fig. 4. A simple example of customizable locking scope determination

In addition, we have further devised a three-level awareness mechanism for distinguishing the working region (locked and highlighted), strongly-depended regions (locked and highlighted), and weakly-depended region (unlocked but also highlighted). Programmers can intuitively observe and assess the risk of semantic conflicts based on the awareness highlight of a basic region. These three types of regions correspond to different semantic conflict risk. Based on the risk level of semantic conflicts, the working region, strongly-depended regions, and weakly-depended regions are highlighted by a series of colors from darker to lighter (e.g.,

dark blue, normal blue, and light blue). As shown in the Fig. 5, when F-depth = 1 and M-depth = 2, the regions enclosed by the dotted line are locked and the darker colors are applied (the color of the working region is the darkest), and the regions outside the dotted line are the weakly-depended regions with the lighter color.

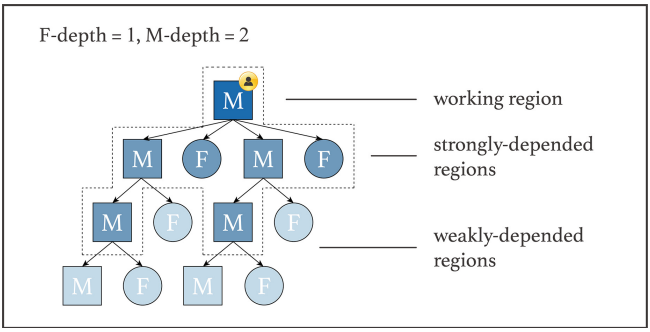

Fig. 5. Example of HSCP scheme

As mentioned, weakly-depended regions are allowed to be edited by others, but working on these regions may still raise semantic conflicts. Therefore, under the HSCP coloring scheme, there are two special cases: (1) whenever a basic region is currently a W-D region (i.e., an overlapping region of working region and depended region by different users), it is highlighted by red; and (2) whenever a basic region is a D-D region (an overlapping region of depended regions from different users), it is highlighted by green. In the HSCP three-level awareness mechanism, the example in Fig. 4 will be turn out to Fig. 6.

5 Major Technical Issues and Solutions

In this section, we introduce technical details of the HSCP scheme. To implement the HSCP scheme on different platforms, we have designed a set of common functionalities that can be reused and integrated into platform-specific collaboration clients. In Sect. 6, we will demonstrate the HSCP prototype implementation respectively on the Eclipse and IDEA platforms.

5.1 Customizable Locking Scope Determination

There are two major data structures in the HSCP scheme: the region state table and the user table. The user table stores all users' latest information for deriving right depended regions. The region state table stores all regions' state information. The function of the region state table is (1) to display the right awareness highlights of the regions (2) to judge whether the local user's editing operations should be allowed.

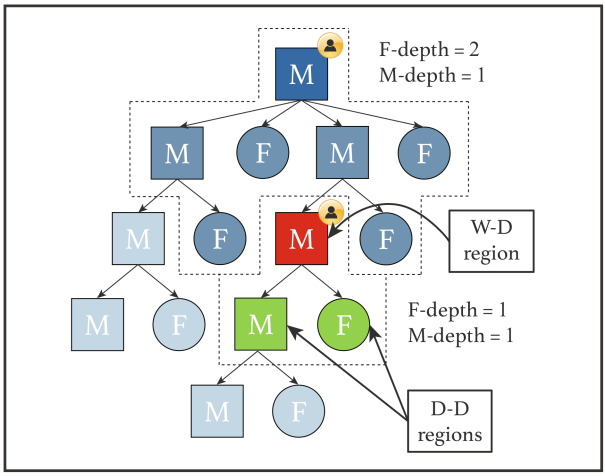

Fig. 6. Two users working under the HSCP three-level awareness mechanism

To implement the customizable locking scope determination, we have added a locking scope policy setting for each user. Each user can configure the local policy setting at any time. In the prior DAL scheme, the *Local Operation Processor* (LOP) and the *Remote Operation Processor* (ROP) have been proposed. For each local user's editing operation, the LOP firstly checks whether the user's operation should be allowed, and then processes the permitted operation, and finally, the operation will be sent to the server for real-time propagation to other sites. For each local and remote editing operation, the region state table will be updated, and the user's locks will be added to all depended basic regions.

Under the HSCP scheme, each user site has a user table that stores all users' latest editing operations and locking scope policy settings. The *CheckPermission* procedure determines the permission for each local editing operation. For each permitted local editing operation, the local locking scope policy setting will be loaded for granting locks with the correct locking scope, and then the policy setting will be sent to the server together with the editing operation. When the ROP receives a message from the server, it retrieves the remote user's editing operation together with the corresponding locking scope policy setting to grant correct locks for the remote user. All editing operations which are permitted in the LOP will not be checked again after being propagated to remote sites. Since each user can receive the policy setting of other users along with the editing operations and all editing operations will only be checked once at their original sites, the global locking state consistency can be ensured by nature.

In order to determine the correct strongly-depended regions according to the user's policy setting, we have re-designed the *DeriveDependedRegionSet* utility function as follows.

– *DeriveDependedRegionSet(R, PS)*: to retrieve the basic regions (as a set) with respect to the user's locking scope policy setting PS and the given source code

region R. In particular, if R contains no strongly-depended region, an empty set will be returned.

The *DeriveDependedRegionSet(R,PS)* function firstly checks the type of the basic regions R. If it is a field, then an empty set will be returned; otherwise, if it is a method, the source code document will be parsed to DG, and then the depended regions within the locking scope according to the F-depth and M-depth of policy setting PS will be added to the result set.

5.2 HSCP Three-Level Awareness Mechanism

Under the HSCP scheme, users can simply and intuitively distinguish different semantic conflict risk of regions based on the awareness highlight.

To retrieve the correct weakly-depended regions, we propose a new utility function named *DeriveAwarenessRegionSet* as shown below. Correspondingly, an *awareness flag* has been proposed. The awareness flags will not be involved in the *CheckPermission* procedure [7], and therefore the local user's editing operations in the regions which are granted awareness flags will not be rejected. The only use of the awareness flags are to provide the correct color of the targeted region.

– *DeriveAwarenessRegionSet(R,PS)*: to retrieve the basic regions (as a set) outside the user's locking scope according to policy setting PS with respect to the given source code region R. In particular, if R has no weakly-depended region, an empty set will be returned.

The *DerieveAwarenessRegionSet(R,PS)* function firstly checks the type of the basic regions R. If it is a field, then an empty set will be returned; otherwise, if it is a method, the source code document will be parsed to a DG, and then all depended regions outside the locking scope according to the F-depth and M-depth of policy setting PS will be added to the result set.

Correspondingly, we have re-defined the *CFD_LSU* utility function as follows. The CFD_LSU is invoked after each local or remote editing operation has been executed.

To retrieve the highlight color of a basic region according to the region state table, a utility function *GetHSCPAwareness* is devised as follows.

– *GetHSCPAwareness(R)*: to retrieve the highlight color of basic region R. If there is only one lock or awareness flag associated with R, return the corresponding color of the owner. If there are multiple locks or awareness flags in the basic region R, when there are any working region locks, return a W-D color (red); otherwise, return a D-D color (green).

5.3 HSCP Implementation: Architecture and Components

Under the HSCP scheme, each collaborating site maintains a user table and a region state table as shown in Fig. 7 below. The HSCP user table is used to store

Algorithm 1. CFD_LSU(O)

```
1: for each site i in the session do
2:     ReleaseLocks(i);
3:     Contextualize(O, i);
4:     W := DetectTargetedRegion(EP(i));
5:     if W != OA then
6:         DRS := DeriveDependedRegionSet(W, PS(i));
7:         DARS := DeriveAwarenessRegionSet(W, PS(i));
8:         GrantLocks(i, W, DRS);
9:         GrantAwarenessFlags(i, DARS);
10:    end if;
11: end for;
```

the latest editing positions and locking scope policy settings of all collaborators and the region state table is used to record the locks and awareness flags of all basic regions in the current document.

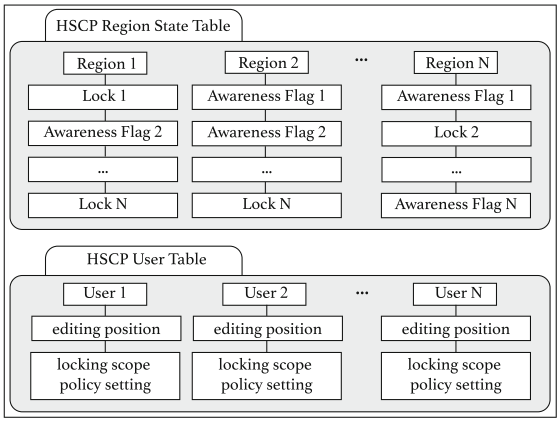

Fig. 7. The major data structures of the HSCP scheme

In software development processes, programmers may prefer different programming IDEs. To implement the HSCP scheme on different platforms (IDEs) without redundant work, we have designed an architecture that has separated a set of common functionalities from platform-specific client components. The overview of the HSCP architecture is presented in Fig. 8. The HSCP Core Module provides common functionalities, while the *HSCP Cross-platform Interfaces* define platform-specific components. Under this architecture, dependency-related information can be retrieved by implementing the platform-specific interfaces. After being processed by the *HSCP Core Module*, the information will be transmitted to other collaborators and stored in the local *HSCP Data Structures*.

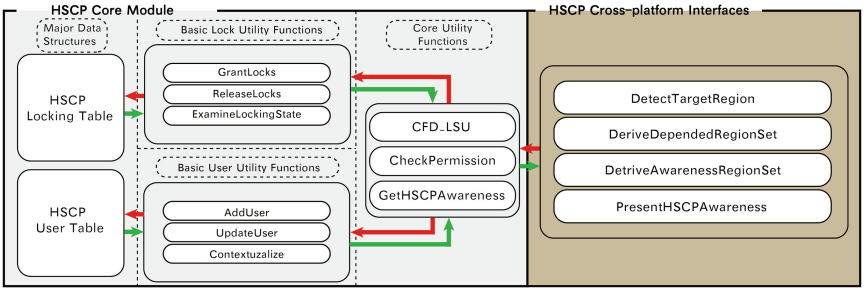

Fig. 8. The overview architecture of HSCP scheme

6 Prototype Implementation and Evaluations

CoEclipse and CoIDEA are two real-time collaborative programming systems proposed and implemented in prior work, which transparently converted the single-user Eclipse and IntelliJ IDEA IDEs into multi-user real-time collaborative programming tools [3]. In this study, we have successfully implemented the HSCP scheme and integrated it into the two prototype systems. In this section, we demonstrate the prototype systems, preliminary user evaluations, and a comprehensive set of performance evaluations.

6.1 Major User Interfaces of HSCP Prototype System

In this section, we demonstrate the HSCP prototype system used in a real-time collaborative programming process.

As presented in Fig. 9, User A on the left is using the Eclipse IDE, setting F-depth as 1 and M-depth as 0; and User B on the right is using IntelliJ IDEA, setting F-depth as 1 and M-depth as 2. User A and User B are editing the source code of *Queue.java*. User A is editing method *offer*, and the awareness highlights are immediately displayed at User B's site. It is worth mentioning that User A's working and depended regions are currently highlighted with three colors: the working region (method *offer*) is highlighted by dark blue, the strongly-depended region (field *rear*) is highlighted by blue, and the weakly-depended region (method *isFull*) is highlighted by light blue.

When User B is editing method *poll*, B's locks and awareness highlights will immediately appear on User A's site as shown in Fig. 10. The M-depth of B is 2, and the method *isEmpty* is a strongly-depended region. It can be seen from the figure that there are three fields (field *front*, *capacity*, and *array*) which are overlapping depended regions, and presenting the D-D color (green). From the figure, we can intuitively observe the risk of semantic conflicts in different regions based on the highlight colors. If User B clicks on User A's working region, a rejection notification will pop up, and the system prevents User A from working in this basic region.

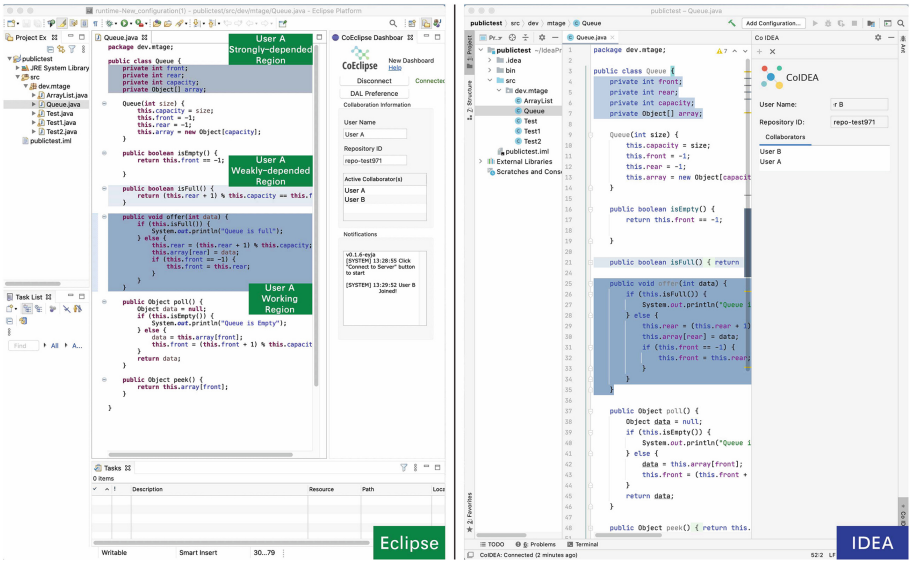

Fig. 9. The three-level awareness of HSCP scheme

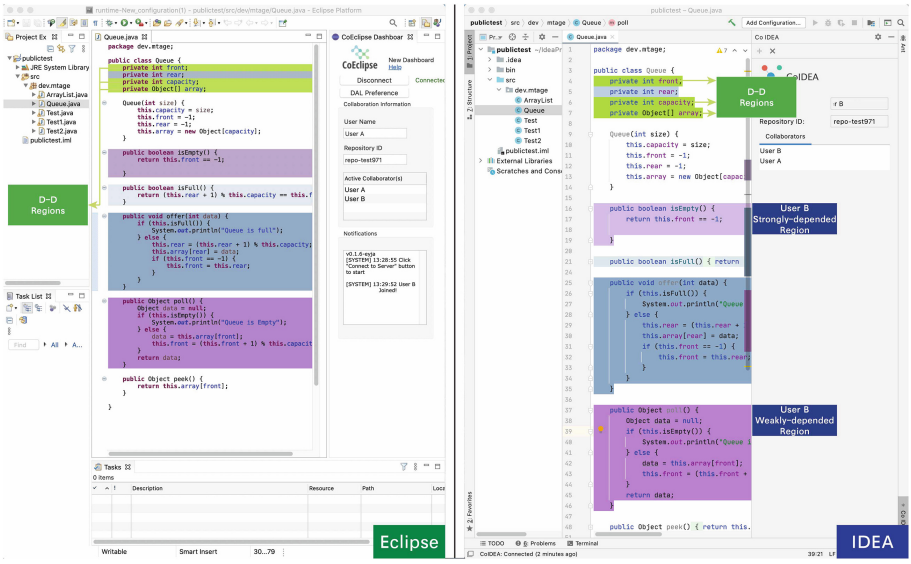

Fig. 10. Programming concurrently under the HSCP scheme

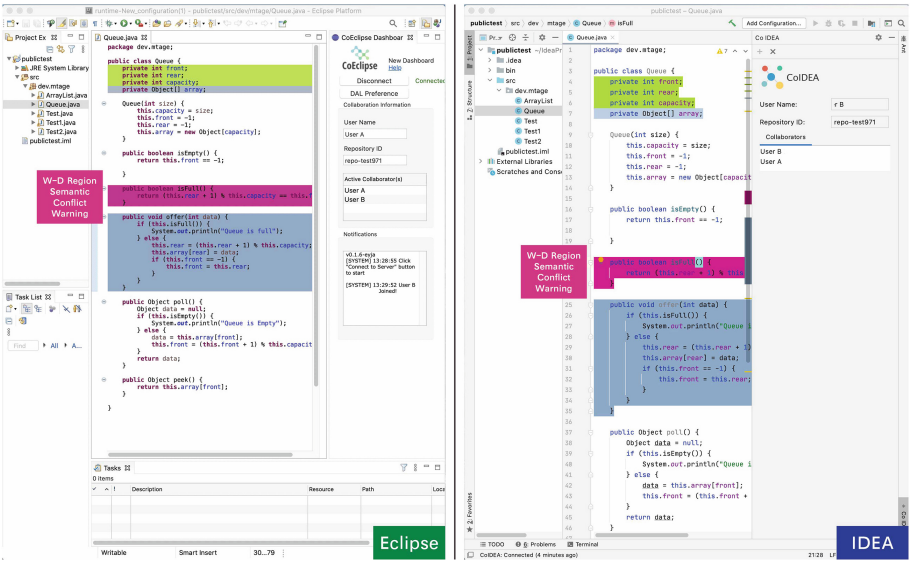

Fig. 11. Special warning of the risk of semantic conflict

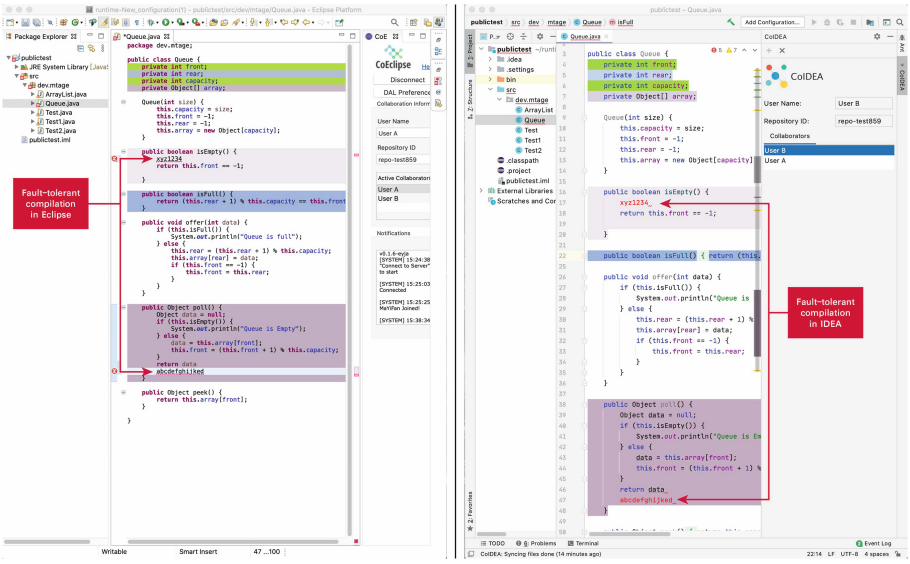

Fig. 12. Fault-tolerant source code analysis mechanism in supporting DAL

When User B is ready to work in method *isFull* which is a weakly-depended region of User A, there will be no rejection, but the region will be highlighted by the W-D color (red) as a special warning, as shown in Fig. 11.

It is worth pointing out that the underlying source code analysis mechanism (for deriving the DG, i.e. basic regions and dependency relationships) has the capability of fault-tolerant parsing. Given any uncompleted code which is syntactically invalid, the above mechanism can always generate basic regions and dependency relationships as much as possible. For example, as illustrated in Fig. 12, there are syntactical errors in both the working region and the depended region, but the DAL mechanism still behaves correctly (with correct highlight awareness and locks).

6.2 Preliminary User Evaluations

The prototype implementation has validated the feasibility of the proposed HSCP approach and techniques. A group of programmers participated in preliminary user evaluations (Fig. 13 illustrates a picture taken during the experiment) and provided positive feedback on semantic conflict prevention features based on the HSCP scheme. Firstly, the programmers selected basic regions, and the HSCP awareness could be immediately displayed on all collaborators' screens. Secondly, when programmers modified the DG structure, the HSCP awareness of all programmers also changed accordingly. Thirdly, when a programmer selected the region locked by others, a UI notification will pop up and the programmer's editing operation were rejected. Finally, when a programmer works in another collaborator's depended region, the highlight of the region became the W-D color to indicate the risk of semantic conflict. During the collaboration process, the syntactic consistency of the shared source code has always been maintained, based on the implementation of the OT algorithm. In addition, we also invited several groups of students to use the prototype system in their programming projects, and received positive feedback as well.

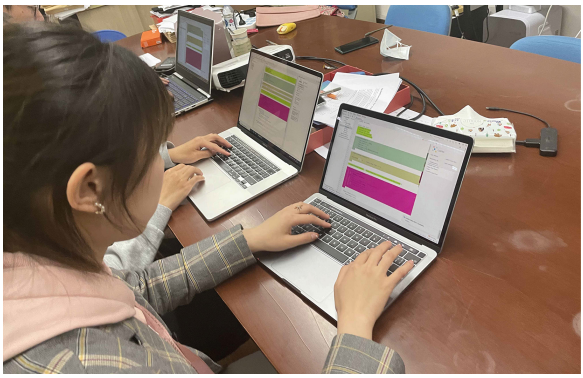

Fig. 13. A picture taken in a laboratory at Tongji University

6.3 Experimental Evaluations

In addition to preliminary user evaluations, we further conducted a comprehensive set of performance evaluations. We evaluated the HSCP core module and the cross-platform interface implementations respectively on CoEclipse and CoIDEA platforms.

Under the HSCP scheme, the amount of basic regions and dependency relationships in a source code document is a key factor that can determinate the performance of the HSCP mechanism. In order to systematically and comprehensively evaluate the performance of the HSCP system, we have developed a utility tool that can generate source code documents based on customizable parameters such as the amount of basic regions and dependency relationships. With multiple documents as input, a comprehensive set of performance evaluation can be conducted. The experimental platform is a laptop with a processor of Intel Core i5-7200U @ 2.5 GHz and 8 GB of RAM.

Firstly, we have evaluated the performance of the implementations of platform-specific components *DetectTargetedRegion*, *DeriveDependedRegionSet* and *DeriveAwarenessRegionSet* on the CoEclipse and CoIDEA platforms, respectively. The performance of these three utility functions depends on the amount of basic regions (denoted as N) and the amount of dependency relationships (denoted as M). Due to the different implementations of AST analyzers on different platforms, the performance of these functions on different platforms is also different. Table 2 and Table 3 present the evaluation results of these utility functions on the CoEclipse and CoIDEA platforms, respectively. The F-depth and M-depth of the customizable policy setting are both 3. As presented, the execution time in the two tables grows steadily with the increase of N and M values. In these utility functions, because *DeriveAwarenessRegionSet* needs to traverse all dependency relationships, it takes the longest time. When the N is 800 and the M is 32000, which is almost impossible in a real scenario, the *DeriveAwarenessRegionSet* function costs 9.84124×10^{-2} s on the CoIDEA platform and costs 2.511987×10^{-1} s on the CoEclipse platform.

Table 2. Performance evaluation of HSCP platform-specific components on the CoEclipse platform

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
DetectTargetRegion	0.044 ms	0.098 ms	0.228 ms	0.237 ms	0.525 ms
DeriveDependedRegionSet	7.289 ms	10.668 ms	16.633 ms	24.435 ms	56.553 ms
DeriveAwarenessRegionSet	7.409 ms	17.128 ms	46.291 ms	92.605 ms	251.199 ms

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

Secondly, we have evaluated the performance of major procedures within the core module of the HSCP scheme, namely *CFD_LSU* and *CheckPermission*.

Table 3. Performance evaluation of HSCP platform-specific components on the CoIDEA platform

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
DetectTargetRegion	0.002 ms	0.002 ms	0.003 ms	0.003 ms	0.004 ms
DeriveDependedRegionSet	0.241 ms	0.428 ms	0.511 ms	0.643 ms	0.699 ms
DeriveAwarenessRegionSet	5.830 ms	11.545 ms	24.022 ms	46.077 ms	98.412 ms

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

The performance of these utility functions also depends on the amount of collaborating programmers (denoted as P). Accordingly, by specifying the amount of collaborating programmers (P), the amount of basic regions (N), and the amount of dependency relationships (M), 20 groups of experiments for each procedure have been conducted. Since we are evaluating the platform-independent core module, we ignore the time spent on platform-specific components. Table 4 presents the experimental results, which have confirmed the good performance of the core module. As presented, the execution time grows steadily with the increase of N, M and P values. When the N is 800, the M is 32000 and the P is 16, which is almost impossible in a real scenario, the *CheckPermission* function costs only 2.3658×10^{-3} s and *CFD_LSU* function costs only 6.3966×10^{-2} s.

Table 4. Performance evaluation of the HSCP core module

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	*CheckPermission*	0.005 ms	0.021 ms	0.065 ms	0.111 ms	0.347 ms
	CFD_LSU	0.063 ms	0.195 ms	0.654 ms	2.549 ms	7.312 ms
4	*CheckPermission*	0.012 ms	0.022 ms	0.099 ms	0.214 ms	0.661 ms
	CFD_LSU	0.245 ms	0.686 ms	2.396 ms	5.179 ms	17.206 ms
8	*CheckPermission*	0.027 ms	0.031 ms	0.180 ms	0.665 ms	1.232 ms
	CFD_LSU	0.897 ms	2.786 ms	7.526 ms	8.415 ms	35.449 ms
16	*CheckPermission*	0.028 ms	0.037 ms	0.208 ms	1.151 ms	2.366 ms
	CFD_LSU	4.009 ms	8.995 ms	17.668 ms	25.129 ms	63.966 ms

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

7 Conclusions and Future Work

Dependency-based automatic locking (DAL) is an approach for supporting semantic conflict prevention in real-time collaborative programming. However, the prior DAL scheme produces extremely and unnecessarily large locking scope, which seriously impacts the concurrent programming work. To better balance

concurrent work and semantic conflict prevention, we have proposed a novel Hybrid Semantic Conflict Prevention (HSCP), which combines locking and awareness approaches. Locking is enforced on the working regions and strongly-depended regions, whereas awareness highlight is applied on weakly-depended regions. Each programmer can customize the depth of the locking scope freely with a fine-grained locking scope policy setting. In addition, a three-level awareness mechanism has been devised, and the conflict risk on different regions can be intuitively distinguished by collaborating programmers. Compared with previous DAL schemes, this work provides users with customization options, and improves the flexibility of semantic conflict prevention.

For implementing the HSCP scheme on different platforms, we have designed a system architecture that separates HSCP's common functionalities from platform-specific components. As examples, we have successfully implemented the HSCP scheme and functionalities on two prototype systems named CoEclipse and CoIDEA, which support programmers to conduct real-time collaborative programming work while enjoying semantic conflict prevention over Eclipse and IntelliJ IDEA. A comprehensive set of experiments have confirmed the responsiveness, efficiency, effectiveness and usability of the HSCP scheme and solution in real-world scenarios.

We have been continuously working in the field of semantic conflict prevention for real-time collaborative programming, and our future work includes (a) supporting multiple locking groups (with multiple working regions) for an individual programmer in the session; and (b) achieving semantic conflict prevention across multiple source code documents. We are continuously developing and improving the research prototype, and planning to release the programs and source code for the community to utilize, with more in-depth evaluations.

Acknowledgment. This study has been sponsored by the Natural Science Foundation of Shanghai (No. 21ZR1465100), the National Natural Science Foundation of China (No. 61772371, No. 62172301, No. 62173248, No. 62073245, and No. 61702374), and the Fundamental Research Funds for the Central Universities.

References

1. Chen, Y., Lee, S.W., Xie, Y., Yang, Y., Lasecki, W.S., Oney, S.: Codeon: on-demand software development assistance. In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, CHI 2017, pp. 6220–6231 Association for Computing Machinery, New York (2017)
2. Fan, H., et al.: CoVSCode: a novel real-time collaborative programming environment for lightweight ide. Appl. Sci. **9**(21), 4642 (2019)
3. Fan, H., Sun, C.: Achieving integrated consistency maintenance and awareness in real-time collaborative programming environments: the CoEclipse approach. In: Proceedings of the 2012 IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 94–101 (2012)

4. Fan, H., Sun, C.: Dependency-based automatic locking for semantic conflict prevention in real-time collaborative programming. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC 2012, pp. 737–742. Association for Computing Machinery, New York (2012)
5. Fan, H., Sun, C., Shen, H.: ATCoPE: any-time collaborative programming environment for seamless integration of real-time and non-real-time teamwork in software development. In: Proceedings of the 17th ACM International Conference on Supporting Group Work. p. 107–116. GROUP '12, Association for Computing Machinery, New York (2012)
6. Fan, H., Sun, C., Shen, H.: ATCoPE: any-time collaborative programming environment for seamless integration of real-time and non-real-time teamwork in software development. In: Proceedings of the 17th ACM International Conference on Supporting Group Work, pp. 107–116 (2012)
7. Fan, H., Zhu, H., Liu, Q., Shi, Y., Sun, C.: A novel dal scheme with shared-locking for semantic conflict prevention in unconstrained real-time collaborative programming. IEEE Access **5**, 22566–22583 (2017)
8. Feldman, M.: CodeSync: a collaborative coding environment for novice web developers. Wellesley College, Wellesley (2014)
9. Goldman, M., Little, G., Miller, R.C.: Real-time collaborative coding in a web IDE. In: Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology, pp. 155–164 (2011)
10. Guo, P.J., White, J., Zanelatto, R.: Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In: 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 79–87. IEEE (2015)
11. Kurniawan, A., Soesanto, C., Wijaya, J.E.C.: Coder: real-time code editor application for collaborative programming. Procedia Comput. Sci. **59**, 510–519 (2015)
12. Lautamäki, J., Nieminen, A., Koskinen, J., Aho, T., Mikkonen, T., Englund, M.: Cored: browser-based collaborative real-time editor for java web applications. In: Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work, pp. 1307–1316 (2012)
13. Nguyen, V., Dang, H.H., Do, N.K., Tran, D.T.: Enhancing team collaboration through integrating social interactions in a web-based development environment. Comput. Appl. Eng. Educ. **24**(4), 529–545 (2016)
14. Sun, C.: OTFAQ: Operational Transformation Frequently Asked Questions and Answers. Nanyang Technological University (2015)
15. Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D.: Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Trans. Comput. Hum. Interact. **5**(1), 63–108 (1998)
16. Wang, Y., Wagstrom, P., Duesterwald, E., Redmiles, D.: New opportunities for extracting insights from cloud based IDEs. In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 408–411 (2014)
17. Warner, J., Guo, P.J.: CodePilot: scaffolding end-to-end collaborative software development for novice programmers. In: Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, pp. 1136–1141 (2017)