An Inconsistency Management Support System for Collaborative Software Development

Phan Thi Thanh HUYEN†a, Student Member and Koichiro OCHIMIZU†, Nonmember

SUMMARY In collaborative software developments, many change processes implementing change requests are executed concurrently by different workers. The fact that the workers do not have sufficient information about the others’ work and complicated dependencies among artifacts can lead to unexpected inconsistencies among the artifacts impacted by the changes. Most previous studies concentrated only on concurrent changes and considered them separately. However, even when the changes are not concurrent, inconsistencies may still happen if a worker does not recognize the impact of the changes made by other workers on his changes or the impact of his changes on other workers’ changes. In addition, the changes in a change process are related to each other through their common target of realizing the change request and the dependencies among the changed artifacts. Therefore, to handle inconsistencies more effectively, we concentrate on both concurrent and non-concurrent changes, and the context of a change, i.e., the change process containing the change, rather than the ongoing changes only. In this paper, we present an inconsistency awareness mechanism and a Change Support Workflow Management System (CSWMS) that realizes this mechanism. By monitoring the progress of the change processes and the ongoing changes in the client workspaces, CSWMS can notify the workers of a (potential) inconsistency in advance along with the context of the inconsistency, that is, the changes causing the inconsistency and the change processes containing these changes. Based on the information provided by CSWMS, the workers can detect and resolve inconsistencies more easily and quickly. Therefore, our research can contribute to building a safer and more efficient collaborative software development environment.

key words: inconsistency awareness, context awareness, patterns of inconsistency, change process, collaborative software development

1. Introduction

In collaborative software development, software artifacts, such as source code, with complex dependency relationships are created through the collaboration of many workers. When a worker makes a change to an artifact, this change may affect the artifacts connected to this artifact by dependency, a relationship between two artifacts in which a change to one artifact may affect the other. However, workers do not always have sufficient information on the changes of the other workers. As a result of that, inconsistencies may occur. We define inconsistency as a situation in which some artifacts are assigned values that are different from the intention of a worker because he is unaware of the changes or the impact of the changes made by other workers, to the artifacts to which his changes apply.

Version control systems (VCSs) [2] are indispensable in software development environments because of their support for collaboration, parallel development, and global software development. However, using VCSs leads to a form of workspace isolation [3] in which the workers could be aware of the changes of the others only if they check-in their changes or synchronize their workspaces with the remote repository. Therefore, conflicts, a type of inconsistency arising due to concurrent changes to the same artifact (direct conflict) or to dependency-related artifacts (indirect conflict) [3], are detected after the workers have finished their changes and committed them to the repository. To detect the conflicts earlier when the changes are in progress, recent studies [3]–[9] have concentrated on workspace awareness techniques that collect information about the ongoing changes of the workers, share this information across their workspaces, and alert them of the emerging conflicts.

The previous works notified the workers of the concurrent changes that may cause a conflict. However, even when the changes are not concurrent, inconsistencies may still happen if a worker does not recognize the impact of the previous changes, made by others workers, on his ongoing changes or the impact of his ongoing changes on the previous changes, because of the large amount of related artifacts and the complicated dependencies among them. Figure 1 shows a simplified situation to illustrate this problem. Mary and Tom change artifacts $D_1$ and $D_2$ respectively at different times. First, Mary checks-out a working copy of the remote repository to her workspace, modifies $D_1$, and checks-in her changes successfully. Later, Tom wants to change $D_2$. We assume that he has changed many artifacts but not $D_1$ since the last commit. When he synchronizes his workspace with the repository before starting modifying $D_2$, VCS tells him that his merge on $D_1$ is clean. As a result, he just focuses on other merge conflicts and may overlook Mary’s change of $D_1$. However, $D_1$ depends on $D_2$. Modifying $D_2$ without considering the change history of $D_1$ may lead to an inconsistency relating with the newly generated version v2.4.
of D2 and the version v1.2 of D1. Furthermore, in the previous works, changes to artifacts are considered individually. However, due to the dependencies among software artifacts, a change to an artifact may affect many other artifacts. This means that, a worker often needs a change process that is a sequence of tasks applying changes to many artifacts, to fulfill a change request. The changes in a change process are not separated but related to each other through their common target of realizing the change request, and the dependencies among the artifacts affected by the changes. Therefore, ignoring the context of a change, the change process containing the change, may lead to some unexpected inconsistencies that may only be detected much later during the build process, integration test, or even runtime failure. Also, to resolve an inconsistency, the workers need to consider the contexts of the changes causing the inconsistency, rather than just the changes themselves. Nevertheless, remembering all the changes one has made or investigating the change history of other workers by oneself is not easy, especially with complex change requests.

In [1], we have proposed an approach that deals with the above problems by representing the change processes in the system explicitly as Change Support Workflows (CSWs) and managing their execution. A CSW is a sequence of activities defined to carry out a change request. Regarding inconsistency awareness, namely recognizing the existence of an inconsistency, most previous works concentrated only on conflicts that were classified into direct conflicts and indirect conflicts by [3]. Besides handling the conflicts like the previous works, we also pay attention to other types of inconsistency by considering the non-concurrent changes and the contexts of the changes that are the CSWs containing the changes. Table 1 shows our patterns of inconsistency classified with regard to the time orders of the activities applying the changes to the artifacts, concurrent or not, the relationships between the artifacts, same or not, and the contexts of the changes, same CSW or not. Our work in [1] focuses more on the unexpected influences among activities in different CSWs. Each pattern of inconsistency represents an inconsistency problem in collaborative software development. Defining the inconsistency situations in the form of patterns aims to provide a common vocabulary among the workers about the inconsistency situations to help them recognize and resolve the inconsistencies more easily. Also, the patterns themselves are easier to be added, improved, or removed.

Extending our previous work in [1], this paper presents in detail the inconsistency awareness mechanism that combines the workspace awareness technique and the context awareness technique [11] to help detect and resolve inconsistencies more effectively. Context awareness means sharing information about the CSWs of the changes. A CSW provides details of the change history, current changes, and likely changes. Then, we develop a Change Support Workflow Management System (CSWMS) (Fig. 2) that realizes this mechanism. In CSWMS, each worker follows a CSW when he performs a change request. By monitoring the progress of CSWs and the ongoing changes in the client workspaces, CSWMS can notify the workers of a (potential) inconsistency in advance, along with the context of the inconsistency that is the changes causing the inconsistency and the CSWs containing these changes. Showing the workers the context of inconsistency apparently helps them understand and resolve the inconsistency, or skip a wrong alarm of inconsistency, more easily and quickly, compared to the previous works which show the workers the concurrent changes causing a (potential) conflict only. Descriptions of the development and implementation of CSWMS are given. The current prototype of CSWMS allows the workers to define, modify, and execute CSWs easily and to receive the warnings of (potential) inconsistency and their contexts. The successful development of this prototype implies the practicability of our proposed approach, as well as the possibility to develop and deploy a mature version of CSWMS. We expect that our proposed approach and CSWMS can help workers implement changes more safely and efficiently in collaborative environments.

The remainder of the paper is organized as follows. Section 2 presents the inconsistency awareness mechanism. Section 3 describes the development of CSWMS in brief. Section 4 illustrates the effectiveness of our approach through the CSWMS prototype. Section 5 addresses related work. Section 6 discusses future work and concludes the paper.

2. Inconsistency Awareness Mechanism

Our inconsistency awareness mechanism is a combination of the workspace awareness and context awareness techniques. Monitoring the progress of CSWs (context awareness), such as the start and finish of an activity in a CSW, helps obtain the context of a change in the system. Monitoring the changes at client workspaces (workspace aware-
ness), such as renaming an existing method or adding a new method, helps detect a new change immediately. Analyzing the latest information about the changes in the system and their contexts allows us to identify in advance the situations that may lead to the patterns of inconsistency [1]:

- **Direct-Conflict** is an inconsistency in which concurrent activities in different CSWs change the same version of an artifact.
- **Indirect-Conflict** is an inconsistency in which a change to an artifact affects concurrent changes to other artifacts by activities in different CSWs.
- **Direct-Revision-Inconsistency** is an inconsistency in which there are contradictory intentions in revising the same artifact at different times by different workers in different CSWs.
- **Indirect-Revision-Inconsistency** is an inconsistency in which a later change to an artifact affects earlier changes to other artifacts by activities in different CSWs.
- **Interleaving-Inconsistency** is an inconsistency in which there are inconsistent views of using a shared artifact by two activities in the same CSW, because the shared artifact is modified by an activity in another CSW in the interval between these two activities.

### 2.1 Information Preparation for Inconsistency Detection

A CSW is a sequence of activities defined to carry out a change request. The activities in a CSW take care of creating new artifacts, modifying, or deleting existing ones. A CSW contains information about the activities, change orders, artifacts accessed by the change activities, change types, change workers, and the execution times of the change activities. Following is its formal definition that is an extension of the definition we have given in [1].

**Definition 2.1.1:** (Change Support Workflow) A CSW is a tuple \(<id, A, E, D, C, CT, W, GD, GC, GW, GT>\) where:

- **id** is the workflow identifier.
- **A** is a set of change activities.
- **E** \(\subseteq (A \times A)\) is a set of directed edges that represent the orders of the change activities.
- **D** is a set of artifacts accessed by the activities of the CSW.
- **C** is a set of changes applied to artifacts by the activities in the CSW.
- **CT** = \([r, w^a, c^m, w^d]\) is a set of types of change to artifacts (r: read, w: write, a: add, m: modify, d: delete). read means that this artifact is for reference only. write means that this artifact is changed. add, delete, modify are subtypes of write.
- **W** is a set of workers who execute the activities of the CSW.
- **GD** : \(C \rightarrow 2^D\) is a function that returns a set of artifacts read/written by a change.
- **GC** : \(A \rightarrow 2^C\) is a function that returns a set of changes made by an activity.
- **GW** : \(A \rightarrow W\) is a function that returns the worker performing a change activity of the CSW.
- **GT** : \(A \rightarrow R^+ \times R^+\) is a time interval function that returns the Start Time (S) and the Finish Time (F) of an activity. \(R^+\) is the set of positive real numbers. 0 denotes an undecided start time or finish time. The interval between S and F is called Execution Time, \(E = F - S\).

Figure 3 shows the structure of a typical CSW with the basic information to detect (potential) inconsistencies based on the explicit properties of the patterns of inconsistency. CSW is just the name of this CSW. Basically, a CSW is a sequence of activities applying atomic changes to artifacts. A worker may make many atomic changes in his workspace during the execution time \([S(A), F(A)]\) of an activity A in his CSW. S(A) and F(A) denote the start time, S, and finish time, F, of the activity A. An atomic change adds, deletes, or modifies an artifact that is named as the main artifact. In order to detect other types of inconsistency besides the direct inconsistencies, we also identify the artifacts that depend on the main artifact, namely inbound artifacts, and the artifacts on which the main artifact depends, namely outbound artifacts. In short, an atomic change includes the information about the main artifact, inbound artifacts, and outbound artifacts. These artifacts and the dependency relationships are extracted automatically by the **Workspace Wrapper** and **Dependency Analysis** components of our CSWMS, respectively (See Sect. 3.2). Also, as a CSW is associated with a project in a worker’s workspace, we need the whole structure of the project to extract the necessary information for inconsistency analysis.

### 2.2 Inconsistency Detection Procedure

Upon receiving a new atomic change C, the following steps are taken:

**Step 1:** Obtain the following information from C: **artifact** (the main artifact), **inbounds** (the inbound artifact list), **outbounds** (the outbound artifact list), **activity** (the change activity containing C), **cswName** (name of the CSW containing the activity), **projName** (name of the project containing the artifacts), **worker** (who made the change), and **version** (version of the main artifact assigned by a VCS).

**Step 2:** Search the database for the atomic changes C’ applied to the artifacts in the same project but different CSWs with C and made by other workers, i.e. C’.projName = C.projName, C’.cswName ≠ C.cswName, C’.worker ≠ C.worker.
Step 3: Analyze the collected data and report a (potential) inconsistency if one of the following conditions holds:

- **Report a Direct-Conflict** between C and C′ if C′ has the same main artifact as well as the version of the main artifact with C, and C′.activity happens concurrently with C.activity (Fig. 4.a):
  \[(C′.artifact = C.artifact) \land (C′.version ≤ C.version) \land ([C′.activity.finishTime, C′.activity.startTime] \cap [C.activity.startTime, C.activity.finishTime]) ≠ ∅\]

- **Report a potential Indirect-Conflict** between C and C′ if C′ has the main artifact d′ depending on the main artifact d of C or depended by d, and C′.activity happens concurrently with C.activity (Fig. 4.b):
  \[(C′.artifact) \lor (C′.artifact) \land ([C′.activity.startTime, C′.activity.finishTime] \cap [C′.activity.startTime, C′.activity.finishTime]) ≠ ∅\]

- **Report a potential Direct-Revision-Inconsistency** between C and C′ if C′ has the same main artifact d with C, the version of d used by C is not older than the version of d used by C′, and C′.activity happens before C.activity (Fig. 4.c):
  \[(C′.artifact = C.artifact) \land (C′.version ≥ C.version) \land (C′.activity.startTime < C.activity.startTime)\]

- **Report a potential Indirect-Revision-Inconsistency** between C and C′ if C′ has the main artifact d′ depending on the main artifact d of C, and C′.activity happens before C′.activity (Fig. 4.d):
  \[(C′.artifact) \land (C′.activity.finishTime < C.activity.finishTime)\]

- **Potential RWR Interleaving-Inconsistency** (Fig. 4.e) For each artifact d in C.outbounds:
  - Find an atomic change C′ having d as the main artifact. C′.activity happens before C.activity:
    \[(C′.artifact = d) \land (C′.activity.finishTime < C.activity.finishTime)\]
  - Find the latest atomic change C″ that is in the same CSW with C and has the main artifact depending on d. C″.activity happens before C′.activity:
    \[(C″.artifact) \land (C″.activity.finishTime < C′.activity.finishTime)\]

If more than one C′ and C″ in the same CSW are found, select the latest ones only.

3. Development and Implementation of CSWMS

This section describes in brief some important points of the development and implementation of CSWMS. Although [1] has proposed an initial design of CSWMS, we have substantially revised and improved it during the real development of CSWMS to make it address the practical implementation issues.

3.1 Static Model

Figure 5 describes the static model of CSWMS.

- Each change request, CR, is implemented by a CSW, CSW. A complex CR, ComplexCR, can be divided into many CRs.
- A CSW is a sequence of change activities, ChangeActivities, that apply atomic changes, AtomicChanges, to artifacts, Artifacts. A CSW can take part in another more complicated CSW, parent CSW.
- Each edge, Edge, will connect two ChangeActivities.
- An Artifact, can depend on many artifacts, outbound
We develop CSWMS with three main functions: CSW management, workspace monitoring, and inconsistency awareness. The first and the second supply the necessary information for the third to analyze inconsistencies. The first function is implemented by the CSW Editor & Executor, CSW Control Engine, User Management, and CSWMS DB components. The second function is implemented by the Workspace Wrapper component. The third function is implemented by the Inconsistency Analysis and Inconsistency Viewer components. In addition, we have the Dependency Analysis component to support other components.

In collaborative software developments, change processes cannot be known entirely at their beginning and they will also evolve during their execution. This means that, CSWs should be able to be defined and executed quickly, and modified even when they are being executed. Bonita 3.1 [12] is a workflow management system that supports collaborative processes, the workflow processes allowing definition and execution operations just after the processes are created, which satisfy the above requirements of CSWs. Therefore, we develop CSWMS by customizing and extending this open source package with functions relating to artifact management and inconsistency awareness.

CSW Editor & Executor allows a CSWMS user to edit and execute a CSW, or modify it during its execution. It is developed by customizing the Activity Manager and Workflow Graph Editor of Bonita 3.1 Client [12], and using the open source JGraph 5.14.0 [13].

CSW Control Engine controls the execution of CSWs. User Management manages Users and Roles of the system. CSWMS DB contains the information about CSWs in the system, Change requests, Roles, Users, and Inconsistencies. These components are developed by customizing and extending Bonita Engine 3.1 [12] to fit the requirements of CSWMS. For example, new data, such as the atomic changes and inconsistencies, are added to CSWMS DB.

Workspace Wrapper monitors the workspaces of the CSWMS users to send to server the atomic changes that are being made in the workspaces by the users, for inconsistency analysis. Workspace Wrapper is implemented as Eclipse plugins that track the whole structure of the project associated with a CSW to find which software artifacts have been changed during the execution of an activity in the CSW. Using Eclipse JDT, Eclipse AST, and Resource Change Tracking techniques of Eclipse, we can identify which type of change (add, delete, modify, or Subversion Synchronize) happened to a Java element, the main artifact of an atomic change, that can be a package, a compilation unit, a class, a method, or a field. Dependency Analysis is then used to identify the other elements of an atomic change including the inbound artifacts and the outbound artifacts (Fig. 3).

Inconsistency Analysis detects (potential) inconsistencies by analyzing the atomic changes received from Workspace Wrapper and the progress of CSWs involved using the mechanism presented in Sect. 2.2.

Inconsistency Viewer shows the contexts of the inconsistencies reported by the CSWMS server. It is implemented as a component integrated in CSW Editor & Executor and Eclipse plugins. Inconsistency Viewer part in CSW...
Editor & Executor supplies the information about the inconsistency-involved CSWs (Fig. 11-B&F, Fig. 12-D&H). Inconsistency Viewer plugins show the inconsistency warnings directly on Eclipse IDE (Fig. 11-D, Fig. 12-F), and the contents of the changes (source code) causing the inconsistencies (Fig. 11-E, Fig. 12-G), using Eclipse Builders, Markers, Natures, Views, etc.

Dependency Analysis appears on both client and server. It uses DependencyFinder [14], a third party tool for creating a dependency graph from compiled Java code, to analyze the dependencies among software artifacts from the code of the entire project. For example, we apply the functions provided by DependencyFinder to find the inbound artifacts and outbound artifacts of the main artifact in an atomic change. Firstly, we generate the dependency graph of the software system with a NodeFactory. The factory keeps track of the package nodes at the top of the graph and all their subordinate nodes. Individual nodes keep track of their outbound and inbound dependencies. Secondly, we traverse the graph to identify the node in the graph corresponding with the changed artifact using the CodeDependencyCollector class. Finally, we call the node.getOutboundDependencies() and node.getInboundboundDependencies() methods to obtain the list of the outbound artifacts and the list of the inbound artifacts, respectively.

3.3 User Guide

To use CSWMS, a worker must be a CSWMS user, that is he is supplied with an account storing his information such as username, password, real name, and email address. The username may be different from the real name of the worker.

A user must login to CSW Manager to do his tasks (Fig. 7-A). CSW Manager (Fig. 7-B) shows the CSWs of the current user in CSW List. Todo List contains the activities which he is assigned to, but has not executed yet. Executing Activity List shows the activities he is executing.

When users double-click on the name of a CSW in CSW List, the detail of this CSW is shown on a CSW graph editor window (Fig. 7-D). Depending on their CSWRoles in a CSW, they can modify the properties of activities, execute activities, or create new activities, etc. The user creating the CSW is called the creator of the CSW. The creator of a CSW is automatically assigned the admin role of this CSW. The creator can define CSWRoles of the CSW, add other users to the CSW, and assign a user to a CSWRole of the CSW. Only the creator and the users assigned the admin role of a CSW can add, delete, or modify its information. A user can execute or update an activity if his CSWRole is the same as the CSWRole of this activity.

Users can create a new CSW from scratch or as a copy of an existing CSW (collaborative workflow or model workflow [12]), or as an instance of an existing CSW (model workflow). From the menu of CSW Manager, select New CSW to open the New CSW dialog (Fig. 7-C). After filling information in this dialog, the Change Request Specification dialog will appear for the creator of the CSW to describe the change request. He can modify this property later by right-clicking the name of this CSW in CSW List of his CSW Manager, and selecting Change Request Property to open Change Request Specification. The name of the new CSW is added to CSW List. On the CSW graph editor window of a CSW, a user can define the activities and their executing orders. The directed edges connecting the activities denote the executing orders of the connected activities. One does not need to specify all activities of a CSW at the beginning. He can add new activities to the CSW, or modify an existing activity, except the finished activities, during the execution of a CSW. For each activity in the CSW, he must specify its name and its CSWRole. An activity contains information about planned changes, the changes he intends to do in this activity, and executed changes, the changes he makes in his workspace during the execution of the activity. Specifying
planned changes is optional for workers. Regarding executed changes, they will be collected automatically by the Workspace Wrapper component of our system. In short, the mandatory information of a CSW that a worker must provide includes the names, CSW Roles, and executing orders of the activities.

Inconsistency notifications are shown in Potentially Inconsistent CSW List of the CSW Manager window and Eclipse IDE (Fig. 11-C&D, Fig. 12-E&F). Users can ignore an alert. However, it is advisable to examine the context of the inconsistency, including the contents of the changes causing the inconsistency and the CSWs of these changes (Fig. 11-E&F, Fig. 12-G&H) to have a timely decision on solving or skipping it.

3.3.1 Activity Life Cycle

The state of an activity is implemented as a property of the activity, a field in the ChangeActivity table in CSWMS DB. At a certain time, an activity can be in one of the following states: INITIAL (light gray background), READY (light yellow background), EXECUTING (light red background), TERMINATED (light cyan background), and DEAD (blue background) (Fig. 8). INITIAL is the state of an activity waiting for the terminations of its parent activities before being ready to start. READY is the state of an activity ready to start. An activity can be in this state if it has no parent activities or all its parents have terminated successfully. EXECUTING is the state of an activity being executed. TERMINATED is the state of an activity that has been terminated successfully. DEAD is the state of an activity that has been canceled. All its depending activities will be automatically canceled.

Users can change the state of an activity by using CSW Manager and CSW graph editor. If they create a change activity without any parents, its state is READY. Otherwise, its state is INITIAL. If a READY activity is assigned to a user, it will appear in Todo List of this user. If the user wants to start an activity in his Todo List, he will select the activity, right click, and choose the Start activity function. The chosen activity will be moved to Executing Activity List, and its state will be changed to EXECUTING. If the user selects an activity in his Executing Activity List, right clicks, and chooses the Terminate activity function, its state will become TERMINATED. If the function he chooses is Cancel activity, its state will be DEAD.

The current state of an activity can be recognized through its background color. Users can also obtain information about the state of an activity, shown on the tooltip of the activity when they pause the cursor over the activity, or on the status bar of CSW graph editor when they select the activity. (Fig. 7-D).

3.3.2 Cooperation between Eclipse IDE and CSW

CSWMS users use the CSW Manager window to create a new CSW first, and then the CSW graph editor window to compose its change activities, executing orders, etc. In Eclipse IDE, users create a workspace, Eclipse workspace, followed by Java projects inside the workspace. At a time, a user works on a Java project by writing code for it. A CSW is associated with a project in the workspace of a CSWMS user. This association is specified in the New CSW dialog (Fig. 7-C) that appears when a user creates a new CSW by choosing the menu item New CSW from the menu of CSW Manager. After a user starts a change activity in his CSW, all changes that happen in the Java project associated with this CSW, executed changes, are collected automatically by the Workspace Wrapper component, every time the user saves his code. The changes that happen in a Java project during the execution of an activity are considered as the executed changes of this activity. In short, using CSWMS, a user can work on the Eclipse IDE in the same way as before. The extra work he must do is to compose a CSW, describe its corresponding Java project, and notify the system when he starts and finishes an activity in the CSW to help the system identify the activity to which an executed change on the IDE belongs.

In addition, it is advisable for a user to update his workspace with the remote repository when he starts an activity, and to commit his changes when he finishes the activity to avoid serious integration headaches, lost work, or not being able to go back to previous versions, etc. As a change request is implemented by a CSW that may include many activities, each activity in the CSW plays a part in the whole story. We expect that users can decompose the goal of the CSW into many sub-goals, and then design the CSW so that each activity is responsible for one sub-goal. In doing so, each activity can make a complete change with regard to its assigned sub-goal. We name these activities standard activities. The activities mentioned in Sect. 2.2 are the standard activities. However, if one feels that the changes of some activity in his CSW are not good enough to check-in, he could finish the activity without check-in. We name these activities local activities. Until some check-in is done, we treat the local activities as the executing standard activities and their atomic changes as the ongoing changes.

3.3.3 Warnings of Inconsistency

The workers should pay attention to the alerts of the inconsistencies defined in the patterns of inconsistency. Currently, we have identified six patterns of inconsistency: Direct-Conflict, Indirect-Conflict, Direct-Revision-Inconsistency, Indirect-Revision-Inconsistency, RWR Interleaving-Inconsistency, and WWR Interleaving-Inconsistency. Therefore,
CSWMS currently shows six alert types corresponding with the six patterns of inconsistency.

4. How CSWMS Supports Workers in Inconsistency Awareness

4.1 An Illustrating Example

To illustrate the effectiveness of CSWMS, we will show an example of getting benefits from CSWMS (Fig. 9). We use the story described in the motivating example in [1], to contrast a traditional environment with the environment supported by CSWMS.

The story is related to two developers of a hypothetical airline-ticket sale software system, Mary and Tom, and an excerpt from the source code of the system, including four Java files, Customer.java, VIPCustomer.java, RegularCustomer.java, and Display.java. In Fig. 9, the class diagram, obtained by applying reverse engineering to these Java files, shows the latest state of the mentioned files after Mary and Tom apply a sequence of changes to these files. The remaining parts in this figure, 1, 2, 3, 4, explain the sequence and the contents of these changes. We use the class diagram instead of the source code for ease of understanding.

Without CSWMS

1 Mary was assigned to fulfill a change request that shows the accumulated point of a customer. She added the showPoint() signature to the Customer interface. Because both the VIPCustomer and RegularCustomer classes implement this interface, she needed to implement the showPoint() method for these classes. To avoid compilation errors, two empty showPoint() methods were temporarily added to these classes.

2 Next, she implemented the showPoint() method of the VIPCustomer class using the showCustomerScreen() method of the Display class. In the morning, she finished this function. Because she had a meeting, she checked in her changes and delayed implementing the showPoint() method of the RegularCustomer class until that night at home.

3 In the afternoon, Tom, the author of the Display class, decided to distinguish the display screen of VIP customers from that of regular customers by modifying the showCustomerScreen() method, and adding a new method, showVIPCustomerScreen(), to display the screen of VIP customers. He also checked in his changes successfully.

4 At night, Mary checked out the project to her workspace and continued the remaining work with her original thinking about the Display class. She also used the showCustomerScreen() method for implementing the showPoint() method of the RegularCustomer class.

Because there are no syntax conflicts on the changed artifacts and these three changes happened sequentially, VCSs and the workspace awareness techniques do not report any errors in this situation. Unfortunately, there are some semantic inconsistencies here. First, there is an Indirect-Revision-Inconsistency, in which the change of Mary on the showCustomerScreen() method affected the earlier change of Mary on the showPoint() method of the VIPCustomer class, because Mary implemented this method with reference to the showCustomerScreen() method. Another inconsistency is RWR Interleaving-Inconsistency, in which Mary implemented the showPoint() methods for the VIPCustomer and RegularCustomer classes with the same view of the showCustomerScreen() method. However, she did not recognize that Tom had modified this method in the interval between her two changes.

These inconsistencies could have been detected before they propagated further, if Tom had recognized the changes of Mary and notified her of his changes or Mary had recognized the effects of his changes and revised the implementation of the showPoint() method in the VIPCustomer and RegularCustomer classes. However, a worker does not always have sufficient information about the changes of other workers to notify them of the impacts of his changes on their changes or to revise his changes to suit their changes. Even when a worker synchronizes his workspace with the remote repository, he may generally not suspect a new change if the VCSs tell him that its merge is clean. In addition, it is difficult for a worker to remember all the changes he has made before. Furthermore, locking all the involved artifacts is not suitable for long-term change processes. Therefore, it is necessary to have a system like CSWMS that supports the workers to prevent, detect, and resolve the inconsistencies more effectively.

With CSWMS

1 Mary, whose username in CSWMS is admin, used CSW Manager and CSW graph editor to define a CSW ShowPoint for implementing the change request showing the accumulated point of a customer. The ShowPoint CSW has two activities, Implement showPoint() in VIPCustomer and Implement showPoint() in RegularCustomer.

2 She executed the activity Implement showPoint() in VIPCustomer first. Although she did not specify any planned changes, thanks to the Workspace Wrapper plugin in Eclipse, the atomic changes on Mary’s workspace during the execution of the first activity were still sent to the server automatically (executed changes of the first activity in Fig. 10). When the first activity finished, she delayed the
indirect revision inconsistency warning for Tom: In the afternoon, Tom, whose username is admin2, started modifying the Display class by defining the Display CSW with one activity, Distinguish Screen of VIP-Customer from RegularCustomer. He started this activity using CSW Manager, and then modified the Display class with Eclipse IDE. When he was modifying the showCustomerScreen() method (Fig. 11-A), a warning of Indirect-Revision-Inconsistency was shown in CSW Manager and Eclipse IDE of admin2 (Fig. 11-C&D). If he examines the context of the inconsistency, CSWMS will show him the source code of the changes causing the potential inconsistency (Fig. 11-E) and the ShowPoint CSW of Mary (Fig. 11-F). By examining it, Tom could recognize that his change affects the finished work of Mary, the implementation of the showPoint() method in the VIPCustomer class, and may affect her future work, Implement showPoint() in RegularCustomer. If Tom solves the detected inconsistency here, the situation will be different from the scenario of the traditional environment. In other words, other expecting alerts will not appear, as there is no inconsistency at all. However, we want to keep the contents and orders of the changes unchanged in both environments, with and without CSWMS, to contrast their differences. Therefore, we assume that Tom ignored this warning so that we can correspondingly simulate the situation mentioned in the traditional IDE.

Indirect-Revision-Inconsistency and RWR Interleaving-Inconsistency warnings for Mary: At night, when Mary logged-in to CSW Manager (Fig. 12-A), she also received the warning of Indirect-Revision-Inconsistency like Tom. If she solves it, the inconsistencies will stop here. To follow the scenario in the traditional environment, we again assume that Mary ignored this warning, and started the second activity, Implement showPoint() in RegularCustomer (Fig. 12-B). When she was coding the showPoint() method of the RegularCustomer class using Eclipse IDE (Fig. 12-C), another warning of RWR Interleaving-Inconsistency was shown in her CSW Manager and Eclipse IDE (Fig. 12-E&F). She could examine the context of the new alert, including the source code of the atomic changes causing the inconsistency (Fig. 12-G) and the Display CSW of Tom, admin2 (Fig. 12-H). With the supplied context, Mary could recognize the modification of the showCustomerScreen() method and the appearance of the showVIP-CustomerScreen() method. Therefore, she could coordinate with Tom or solve it by herself.

4.2 Discussion

In a traditional IDE, workers work on their own workspaces with the support of VCSs and can recognize the existence of a Direct-Conflict if VCSs show them a warning of merge conflict when they synchronize their workspaces. However, there are several weaknesses of this environment.
Only Direct-Conflicts can be detected.

Direct-Conflicts are detected late after all the changes involved have been finished.

One does not know about the ongoing changes in the workspaces of the others. He could be aware of their newly committed changes only if he synchronizes his workspace with the remote repository. Some VCSs solve this problem by broadcasting check-in notification emails to workers. However, manually analyzing tons of emails to find the potentially-related commits takes workers a lot of time. Also, because the information in the commit mails is limited, they must then investigate the changes they care by themselves. In both cases, because of the large amount of the received updates and artifacts, and the complicated dependencies among software artifacts, a worker could not always recognize the changes or the impacts of the changes of other workers on his changes and vice versa. Similarly, the solution in which the worker must guarantee the consistency of effects by his changes by notifying his change on an artifact to the current and potential users of the artifact by himself is also tricky.

Workspace awareness techniques support traditional IDEs to detect Direct-Conflicts and Indirect-Conflicts earlier when the changes are in progress. However, as we have discussed in Sect. 1 and Sect. 4.1, dealing with the ongoing changes only is not enough. There are many types of inconsistencies relating with the contexts of changes and changes already been committed.

Our inconsistency detection environment, CSWMS, aims to reduce the load of the workers and to handle inconsistencies more effectively.

Workspace awareness techniques support traditional IDEs to detect Direct-Conflicts and Indirect-Conflicts earlier when the changes are in progress. However, as we have discussed in Sect. 1 and Sect. 4.1, dealing with the ongoing changes only is not enough. There are many types of inconsistencies relating with the contexts of changes and changes already been committed.

Our inconsistency detection environment, CSWMS, aims to reduce the load of the workers and to handle inconsistencies more effectively.

- It can detect many types of inconsistency: Direct-Conflict, Indirect-Conflict, Direct-Revision-Inconsistency, Indirect-Revision-Inconsistency, RWR Interleaving-Inconsistency, and WWR Interleaving-Inconsistency. In the example in Sect. 4.1, our CSWMS could detect Indirect-Revision-Inconsistency and RWR Interleaving-Inconsistency, which a traditional IDE and workspace awareness techniques could not.
- Inconsistencies are detected in advance when the changes involved are happening. Also in the example in Sect. 4.1, with our CSWMS, the inconsistency warnings were shown when Tom and Mary were coding in real time.
- Instead of investigating the changes of other workers
to understand the impacts of their changes or to notify them of the impacts of his changes, by himself, a worker only needs to provide some simple information about his CSW. Managing the execution of CSWs, monitoring ongoing changes and committed changes of the workers, analyzing the collected information to detect inconsistencies, and notifying the workers of the possibility of an inconsistency along with its context are handled automatically by our system.

- Providing the contexts of the changes causing an inconsistency helps the workers understand and resolve the inconsistency more effectively compared with showing the changes only.
- The inconsistency detection environment can offer some permanent benefits to collaborative software development. Through the warnings of inconsistency and showing the CSWs of the workers involved, it contributes to increasing the awareness of a worker about the works of his co-workers. This awareness may help the worker make correct changes and avoid unexpected impacts on the changes of other workers. Also, the stored CSWs are useful for the workers when they need to recall the purposes of their past changes for bug fixes or system maintenance and evolution.

Besides the benefits, CSWMS may issue false warnings that are also the weakness of the workspace awareness techniques [3]–[6]. Traditional IDEs also face this problem because most VCSs use unstructured merge (i.e. line-based) with the file-level granularity. A false warning interrupts the workers from their work and takes time to examine. To reduce the side-effects, we apply the fine-grained analysis that captures and analyzes changes at the level of program entity (structured changes), for instance, class, field, and method. Moreover, we supply the workers with the contexts of the changes causing the warning to help them understand and resolve it more easily. Therefore, the cost of examining the false warnings is expected to be much lower than the cost of fixing the defects that are caused by the inconsistencies detected late in the software life cycle.

Using our system, workers can work on the IDE in the same way as a traditional IDE. The extra work they must do is to compose a CSW using CSW Manager and CSW graph editor. Because we have minimized the information the workers must supply, we believe that this burden will become negligible when they get used to the system. In addition, defining CSWs gives some other benefits. Similarly to plans, CSWs give the workers paths to follow and help them understand their work more clearly. They are also a good communication tool for the workers to understand the goal and activities of each other.

To sum up, our inconsistency detection environment helps the workers understand and resolve the inconsistency, or skip it in the case of a false warning, more effectively. In addition, providing information about the change processes of workers makes their collaborations easier. Therefore, we expect that the advantages of this environment compared with the traditional IDE and workspace awareness techniques outweigh its disadvantages.

5. Related Work

Most studies on inconsistency in collaborative environments are about conflicts caused by concurrent changes of different workers. Traditional approach uses VCSs [2] in conjunction with the software development environment to address the problem of concurrent access. However, direct conflicts are detected only after the workers have checked-in their work. To catch conflicts earlier, many workspace awareness techniques were proposed.

To detect pending conflicts rather than potential conflicts, Crystal [7] merges the local repositories of workers in advance after they check-in in their local repositories but not yet commit to the remote repository.

To detect emerging conflicts, Palantir [3] monitors the workspaces of workers to provide the information about ongoing changes, and notifies the involved workers of potential conflicts. CASI [4] uses visualizations to show which source code entities are being changed. CollabVS [5] enriches Visual Studio IDE with both conflict notifications and communication. To reduce false positives, Syde [6] uses a fine-grained change tracking mechanism in which object-oriented systems are modeled as abstract syntax trees, and changes are tree operations. Similarly to Crystal, WeCode [8] computes the presence of merge conflicts, but by continuously integrating the changes in the workspaces of workers into a merge workspace shared among the workers. Regarding UML diagrams or documents, ADAMS [9] is a web-based system that provides a fine-grained artifact management approach and uses traceability information to propagate the change events to the related artifacts.

CSWMS is an extension of those works. In addition to monitoring the workspaces of workers to detect the concurrent changes that may cause a conflict, CSWMS manages the change processes in the system to provide the workers with the contexts of the changes rather than just the changes themselves, to help them detect and resolve not only conflicts but also other types of inconsistency.

Mylyn [10] is a very popular task and application life-cycle management framework for Eclipse. There are some similarities between Mylyn and CSWMS where the worker must also define a CSW, start his assigned activity, and finish it. However, the context of a task in Mylyn is the interaction of a worker with the system’s elements and relations. Also, Mylyn does not pay attention to the inconsistency problem among its users. Although the primary purpose of CSWMS is different from that of Mylyn, CSWMS could become more powerful if we integrate Mylyn into the current system.

CSWMS also distinguishes itself from Bonita because its CSW extends cooperative process defined in [12] with data and atomic changes. Moreover, inconsistency awareness is not mentioned in Bonita, but a key issue of CSWMS.
6. Conclusion and Future Work

To deal with the inconsistency problem in collaborative environments more effectively, we have proposed an approach that explicitly represents the change processes as Change Support Workflows (CSWs), and manages their execution with respect to our patterns of inconsistency [1].

Extending [1], this paper has presented in detail the inconsistency awareness mechanism and the CSWMS system implementing this mechanism. CSWMS monitors both the progress of CSWs and the ongoing changes in the client workspaces to notify the workers in advance of the (potential) inconsistencies along with their contexts, the changes causing the inconsistencies and the CSWs of these changes. Successful development of a prototype of CSWMS implies the feasibility of our proposed approach and the ability to develop a more mature CSWMS. Therefore, our research can contribute to building a safer and more efficient collaborative software development environment.

To put CSWMS into practice, we will continue improving the user interfaces, and identifying other specific cases of the patterns of inconsistency. Integrating Mylyn [10] into CSWMS is also a promising plan. In addition to handling the inconsistencies, by managing the change processes, CSWMS can be improved to help maintain and update software systems effectively, and to increase the reusability of the change processes. In particular, we may conduct further development on the change request management function, or combine the current system with an existing change request management system to completely manage the life cycle of a change request. After developing a mature version of CSWMS, analyzing its impact on real collaborative software development is necessary to evaluate exactly the effectiveness of our research.

References

[1] P.T.T. Huyen and K. Ochimizu, “Toward inconsistency awareness in collaborative software development,” Proc. 18th APSEC, pp.154–162, 2011.
[2] K. Altmanninger, et al., “A survey on model versioning approach,” IJWIS, vol.5, iss. 3, pp.271–304, 2009.
[3] A. Sarma, et al., “Towards supporting awareness of indirect conflicts across software configuration management workspaces,” Proc. ASE’07, pp.94–103, 2007.
[4] F. Servant, et al., “CASI: Preventing indirect conflicts through a live visualization,” Proc. CHASE’10, pp.39–46, 2010.
[5] P. Dewan and R. Hegde, “Semi-synchronous conflict detection and resolution in asynchronous software development,” Proc. ECSCW’07, pp.159–178, 2007.
[6] L. Hattori and M. Lanza, “Syde: A tool for collaborative software development,” Proc. ICSE’10, pp.235–238, 2010.
[7] Y. Brun, et al., “Proactive detection of collaboration conflicts,” Proc. ESEC/FSE, pp.168–178, 2011.
[8] M.L. Guimaraes and A.R. Silva, “Improve early detection of software merge conflicts,” Proc. ICSE, pp.342–352, 2012.
[9] A.D. Lucia, et al., “Fine-grained management of software artefacts: The ADAMS system,” Software Practice and Experience, vol.40, no.11, pp.1007–1034, 2010.
[10] M. Kersten and G.C. Murphy, “Using task context to improve programmer productivity,” Proc. SIGSOFT’06/FSE-14, pp.1–11, 2006.
[11] I. Omoronyia1, et al., “A review of awareness in distributed collaborative software engineering,” Software Practice and Experience, vol.40, pp.1107–1133, 2010.
[12] F. Charoy, et al., “A dynamic workflow management system for coordination of cooperative activities,” Proc. BPM’06, 2006.
[13] JGraph, http://www.jgraph.com
[14] Dependency Finder, http://depfind.sourceforge.net/