Managing Inconsistency among Software Artifacts

by

PHAN Thi Thanh Huyen

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Koichiro Ochimizu

School of Information Science
Japan Advanced Institute of Science and Technology

September 2013

Abstract

In collaborative software development, many change requests, such as adding or modifying features, or fixing bugs, can be implemented by different workers within a time interval. Each worker conducts his own *change process* that is a sequence of tasks applying changes to a set of artifacts. When a worker makes a change to an artifact, this change may affect the artifacts connected to this artifact by dependencies. However, because of communication problems and the complex and changeable nature of software, the workers do not always have enough information about the work of the others. As a result of that, a change to an artifact of a worker may unexpectedly affect the changes to other artifacts of other workers. Therefore, inconsistencies may occur from the affected artifacts.

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. This situation leads to syntactic errors or semantic errors causing unexpected or unintended behaviors of the constructed software system. The goal of this dissertation is to build a Change Support Environment (CSE) that supports the workers in preventing, detecting, and resolving inconsistencies in a collaborative software development environment more effectively.

This dissertation takes into account the problems not addressed by the existing studies in the area of inconsistency awareness. Differently from these studies that concentrated only on concurrent changes and considered them separately, we pay attention to 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. By considering the inconsistency problem under the view of the change processes containing the changes:

1. We have defined the patterns of inconsistency with regard to the relationships between the affected artifacts, the time orders of the tasks applying the changes to the artifacts, and the change processes of the changes.

2. We have proposed a context-based approach to solve the inconsistency problem more effectively. The change processes in collaborative software development are managed to provide the contexts of the changes in the system. Our inconsistency awareness technique combines monitoring the workspaces of the workers (workspace awareness) with managing the progress of the change processes executed in the system (context awareness) to detect in advance (potential) inconsistencies in real time. Information about the context of an inconsistency, the changes causing the inconsistency and their change processes, is provided to help the workers fully comprehend the inconsistency before resolving it.

3. Based on the above approach, we have developed an inconsistency management support system that allows the workers to define, execute, and modify their change processes easily, and to receive inconsistency warnings along with the contexts of the inconsistencies to resolve the inconsistencies in advance.

4. We have also given a formal method for modeling the main behaviors of CSE in Colored Petri Nets (CPN or CP-nets). CPN Tools is used to verify the generated model to detect the patterns of inconsistency. Our method can be applied to model and verify the data flows of other types of workflows.

The novelty of this dissertation lies in addressing the inconsistency problem by considering the contexts of changes, rather than just the changes themselves. By managing the execution of the change processes, monitoring the ongoing changes in the workspaces of the clients, and using the change history stored in the repositories of VCSs and the planned changes specified by the workers, we can detect in advance emerging inconsistencies that are not reported by the previous studies, and provide in detail the context of an inconsistency to help the workers understand the situation and have a timely decision for resolving the inconsistency before its effect goes further.

In summary, changes are inevitable in collaborative software development environments despite their high cost and risk. Our research, developing the model and the environment able to detect the (potential) inconsistencies that the workspace awareness approach can not detect and supply the workers with the contexts of the inconsistencies, can help the workers implement changes more safely and efficiently in collaborative environments.

Acknowledgment

It has been a long and challenging journey from the shape of the first idea until the forming of this dissertation. Besides many tears for disappointment and ambiguity when getting lost in research, there is also a lot of happiness for achieving the desired results, for challenging myself, and for being able to keep hoping. I would not have overcome the obstacles in my road without the support of so many people around me, who help me stand up after failures and make me firm with the road I have chosen.

First and foremost, I offer my sincerest gratitude to my supervisor, Professor Koichiro Ochimizu, who has supported me throughout my study with his knowledge, guidance, and encouragement. Without his consistent help, this dissertation would not have been completed or written.

I would also like to express my thanks to Professor Motoshi Saeki of Tokyo Institute of Technology, Professor Kunihiko Hiraishi, Associate Professor Masato Suzuki, Associate Professor Toshiaki Aoki, and Associate Professor Kazuhiro Ogata of Japan Advanced Institute of Science and Technology, for their valuable comments.

In addition, I would like to thank the Japanese Government (Monbukagakusho: MEXT) Scholarship for financial support during my study in Japan.

Many thanks to numerous friends, especially those at Ochimizu Laboratory for their willingness to participate in challenging discussions and give help to tackle the language barrier in my daily life.

Last but not least, I thank my family, especially my husband, who has endured this long journey with me, for always offering love, support, and understanding.

Phan Thi Thanh Huyen
September 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems must be changed under various circumstances during development and after delivery, such as for new requirements, error correction, and performance improvement. However, software change is not an easy task, especially in a collaborative environment, where software artifacts with very complex dependencies are created through the collaboration of many workers. One common problem in a collaborative work is that a worker misunderstands, or does not recognize or have sufficient information about the changes or the impact of the changes made by other workers, even if they are in the same team and have been explained about the task of each member, because of communication problems, the large amount of related artifacts, the complicated dependencies among the artifacts, and the intangible, complex, and changeable nature of software. As a result of that, a change to an artifact of a worker may unexpectedly affect the changes to other artifacts of other workers. Therefore, inconsistencies may occur from the affected artifacts. Inconsistencies increase in both size and severity along with the increase in the number of workers, software scale, and software development duration. They can lead to cost overruns, project delays, or even project failure.

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. This situation leads to syntactic errors or semantic errors causing unexpected or unintended behaviors of the constructed software system. The goal of this dissertation is to build a Change Support Environment (CSE) that supports the workers in preventing, detecting, and resolving inconsistencies in a collaborative software development environment more effectively.

Several attempts have been made to detect conflicts, a type of inconsistency caused by concurrent changes to the same artifact (direct conflict), or to dependency-related artifacts (indirect conflict) [21]. A traditional approach provided by version control systems (VCSs) [7] is to detect conflicts when the workers commit their changes to the remote repository. Because conflicts in this situation are just detected after changes have been finished, there is a need for detecting conflicts earlier when the changes are being implemented. Recent studies [21], [24], [18], [20], [25], [28], [26] have concentrated on workspace awareness techniques that monitor the ongoing changes in the workspaces of workers, share this information across the workspaces, and notify the workers of emerging conflicts caused by the ongoing changes. A worker who is changing an artifact will receive a warning of a (potential) conflict if another worker is also changing the same artifact, or a warning of a

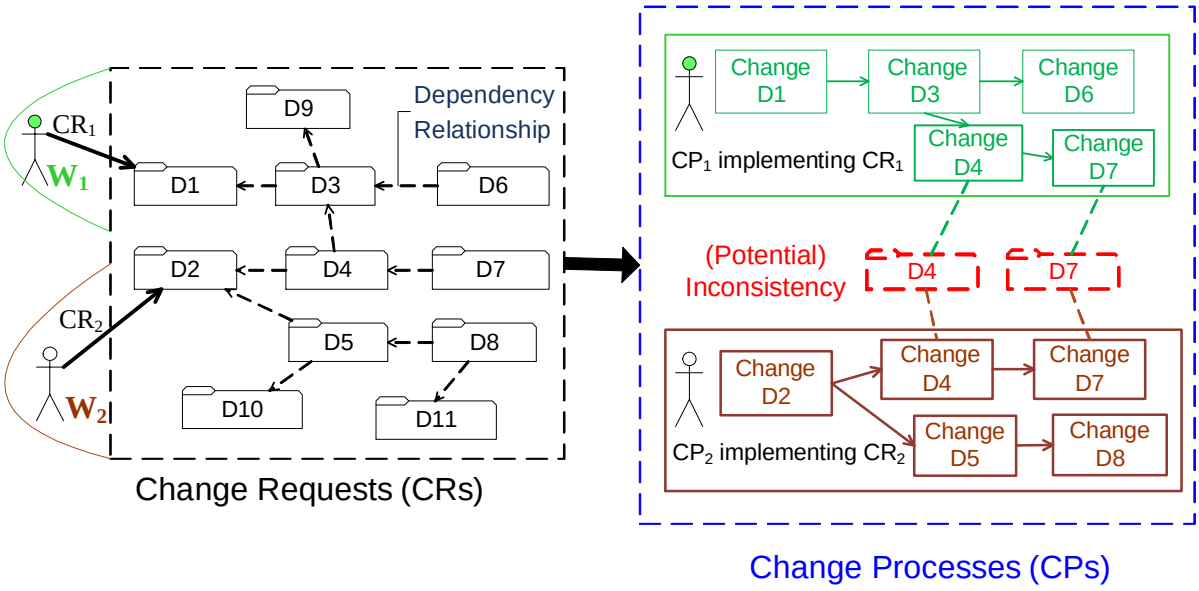


Figure 1.1: Inconsistency in collaborative software development under the view of change processes

(potential) indirect conflict if a concurrent change to another artifact of another worker may affect his ongoing change.

The previous studies concentrated only on concurrent changes that may cause a conflict, and treated the changes separately. However, even if the changes are not concurrent, a worker may not recognize the impact of the previous changes, made before by other workers, on his ongoing changes or the impact of his ongoing changes on the previous changes, because of the large amount of artifacts and the complicated dependencies among them. Moreover, a change to an artifact is often related to a number of changes that share a common target of realizing a change request with this change. In addition, in these studies, when a potential conflict is detected, the awareness of workers is limited to the ongoing changes and related artifacts. Nevertheless, to resolve an inconsistency in general and a conflict in particular, the workers involved may need to reconsider their change history and the planned changes in addition to the ongoing changes. The more information about the inconsistency is provided, the more easily the inconsistency can be solved. Not considering these problems could lead to some unexpected inconsistencies that may only be detected much later in the development process. The later an inconsistency is found, the higher the cost for resolving it is.

Differently from the previous works, the above-mentioned problems are taken into account in this dissertation. We consider the inconsistency problem not at the view of the individual changes but at the view of the context of the change. Because of the dependencies among artifacts, a worker often makes more than one change to many artifacts to implement a change request. We call a sequence of changes to fulfill a change request a *change process*. A change process can reveal much information about a change inside it, such as the preceding changes, the succeeding changes, and the change request realized by the change process. Therefore, we use the change process containing a change to denote the context of the change. From the view of change processes (Fig. 1.1), we:

- Identify and classify the patterns of inconsistency;

- Introduce the theoretical model of CSE that models each change process as a Change Support Workflow;
- Propose an inconsistency awareness technique that detects emerging inconsistencies in real-time by collecting and analyzing the latest information about the ongoing changes before check-in in the workspaces of the workers, in addition to the progress of the change processes in the system;
- Develop an inconsistency management support system based on the above theory;
- Present a method for modeling the main behaviors of CSE in Colored Petri Nets (CPN or CP-nets) and verifying the CPN model to detect the patterns of inconsistency. This method provides an alternative solution suitable for small-size CSEs with CSWs that need to be designed and verified carefully before execution.

The novelty of this dissertation lies in addressing the inconsistency problem by considering the contexts of changes, rather than just the changes themselves. By managing the execution of the change processes, monitoring the ongoing changes in the workspaces of the workers, and using the change history stored in the repositories of VCSs and the planned changes specified by the workers, we can detect in advance emerging inconsistencies that are not reported by the previous studies, and provide in detail the context of an inconsistency to help the workers understand the situation and solve it easily.

In summary, our research can contribute to building a safer and more productive collaborative software development environment by detecting an inconsistency in advance and supplying workers with the context of the inconsistency to help them have a timely decision for resolving the inconsistency before its effect goes further. In addition, managing the change processes occurred during the development of a software system is useful for the maintenance and evolution of the software system in the future.

1.1 Research Contributions

The major contributions of this research are:

- We have defined and categorized the patterns of inconsistency, including the conflict patterns addressed in the previous studies, by considering the relationships between the affected artifacts, same artifact or different artifacts, the time orders of the tasks applying the changes to the artifacts, concurrent or not, and the contexts of the changes that are the change processes containing the changes, in the same change process or in different change processes. Each inconsistency pattern represents an inconsistency problem in collaborative software development. Defining the patterns aims to provide a common vocabulary among the workers about the inconsistency situations to help them recognize and resolve the inconsistencies earlier and more easily.
- We have proposed a context-based approach to deal with the inconsistency problem in a collaborative environment more effectively. By representing the implicit change processes as Change Support Workflows (CSWs) and managing their execution, we can provide information about the contexts of the changes in the system. Regarding *inconsistency awareness*, namely recognizing the existence of an inconsistency,

our technique is a combination of the workspace awareness technique and the context awareness technique. Context awareness means sharing information about the contexts of changes that are the change processes containing the changes. By monitoring the progress of the change processes and the ongoing changes at the client workspaces, we can obtain the latest information of the ongoing changes in the system and their contexts. Therefore, a (potential) inconsistency can be recognized in advance along with the *context of the inconsistency*, that is, the changes causing the inconsistency and the change processes containing these changes. Showing the workers the context of an 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.

- Based on the proposed approach, we have developed a Change Support Workflow Management System (CSWMS) in which each worker follows a CSW to implement a change request. Complete descriptions of CSWMS, including system requirements, static model, dynamic model, and architecture of CSWMS, have been presented. We have also developed a prototype of CSWMS that allows workers to define, execute, and modify CSWs easily, and to receive the warnings of inconsistencies along with the contexts of the inconsistencies to resolve them in advance.

- We have presented a formal model of CSE that formalizes artifacts and both data flow and control flow of CSWs in Colored Petri Nets. CPN Tools is used to edit, simulate, and verify the CPN model to detect data abnormalities, specially the patterns of inconsistency. Successful modeling and verification of CSE are the initial achievements in proving the feasibility and correctness of the approach of CSE. Also, the proposed modeling and verification method, with some advantages compared to the previous works in data-flow verification, can be applied to other types of workflow.

1.2 Dissertation Structure

This dissertation is organized into nine chapters.

- **Chapter 1 - Introduction -** (This chapter)

- **Chapter 2 - Motivation and Approach -** This chapter describes the problems that are not handled by the existing works in inconsistency awareness, and an overview of our approach.

- **Chapter 3 - Patterns of Inconsistency -** We define the patterns of inconsistency with regard to the relationships between the affected artifacts, same or not, the time orders of the tasks applying the changes to the artifacts, concurrent or not, and the contexts of the changes, in the same change process or not.

- **Chapter 4 - Theoretical Model of Change Support Environment -** We describe the theoretical model of Change Support Environment (CSE) that represents the implicit change processes in the system as Change Support Workflows and manages their execution based on the patterns of inconsistency.

- **Chapter 5 - Inconsistency Awareness -** This chapter describes in detail the technique to detect (potential) inconsistencies in real time, and some solutions to the detected inconsistencies. In addition, we present a formal method for modeling the main behaviors of CSE, especially the data aspect, using CP-nets. Then, the generated model is verified to detect the patterns of inconsistency. Because of the modeling cost and the state explosion problem in model checking, this alternative method is suitable for small-size CSEs with important CSWs that take time to design and verify before being executed.

- **Chapter 6 - An Inconsistency Management Support System for Collaborative Software Development -** We present a complete description of the development and implementation of Change Support Workflow Management System, based on the theoretical model of CSE and the technique to detect inconsistency in real time, given in Chapter 4 and Chapter 5.

- **Chapter 7 - Performance Evaluation and Discussion -** We compare our work with the related studies in the field of inconsistency awareness, and then evaluate performance of the inconsistency detection algorithm. We also give some discussions about the effectiveness of our research and the challenges in evaluating its effectiveness.

- **Chapter 8 - Related Work -** This chapter provides an overview of related work and the differences between our research and the related work.

- **Chapter 9 - Conclusion -** This chapter summarizes the major contributions of the dissertation and describes some future directions, especially to overcome the current limitations of our research.

1.3 Summary

In this section, we have introduced an overview of our research. Fig. 1.2 summaries the main points of this dissertation.

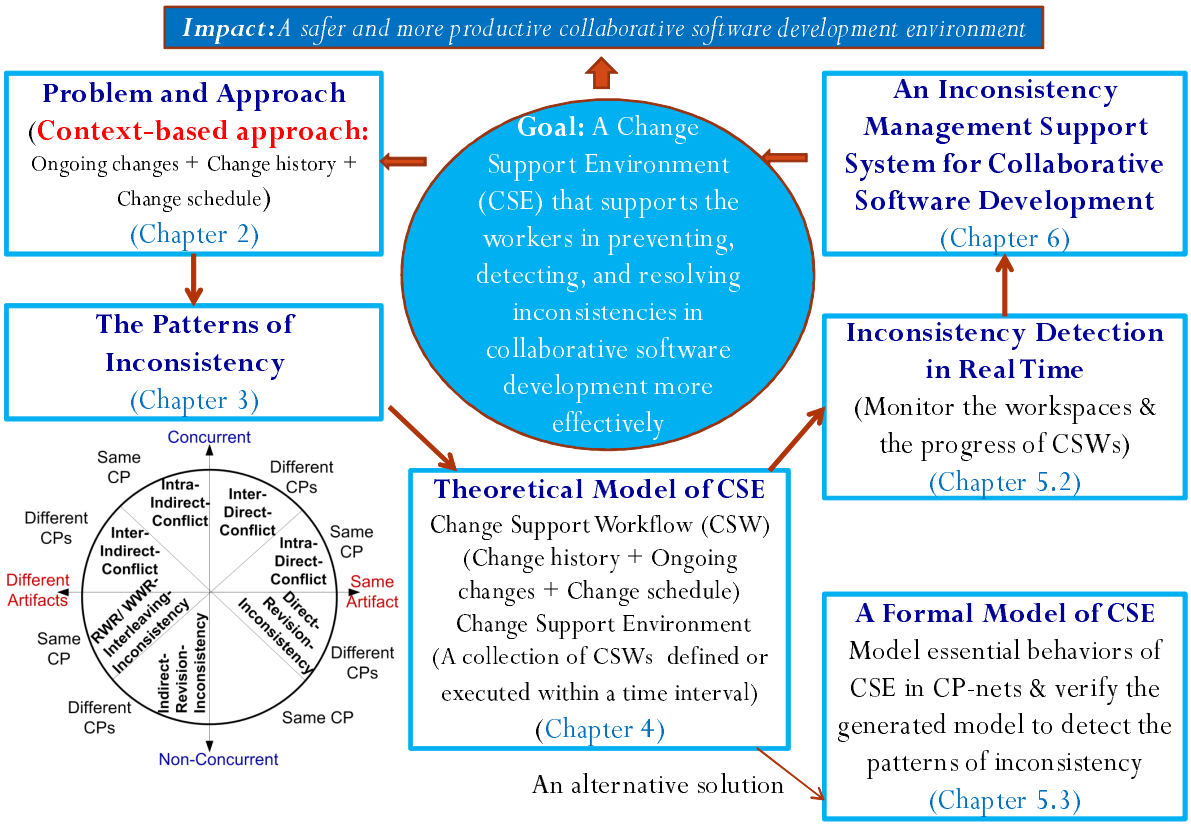


Figure 1.2: Research outline

Chapter 2

Problem and Approach

This chapter presents the motivations and an overview of our approach to managing inconsistencies in collaborative software developments.

2.1 Problem Formulation

Collaboration is indispensable in software development along with the non-stop growth in scale and complexity of software projects. In a collaborative work, software artifacts, such as source codes, with complicated *dependencies* are created through the collaboration of many workers. *Dependency* is a relationship between two artifacts in which a change to one artifact may affect the other. When a worker makes a change to an artifact, this change may affect the artifacts connected to this artifact by dependencies. However, because of communication problems and the intangible, complex, and changeable nature of software, the workers do not always have enough information about the work of the others. These problems are more serious in current practices with parallel development and distributed environment. Therefore, a change of a worker may unexpectedly affect the changes of the others. These coordination breakdowns cause software inconsistencies that, in turn, lead to cost overruns, project delays, or even project failure.

Version control systems (VCSs) [7] are used widely in collaborative software development because of their support for collaboration, parallel development, and global software development. However, using VCSs leads to a form of workspace isolation [21] 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 caused by concurrent changes to the same artifact (direct conflict) or to dependency-related artifacts (indirect conflict) [21], are detected late after the workers have finished their changes. To detect conflicts earlier, when changes are being implemented, recent studies 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. Palantir [21] monitors the workspaces of workers to provide information of ongoing change, and notifies the involved workers of potential conflicts. CASI [24] uses visualizations to show which source code entities are being changed. CollabVS [20] enriches Visual Studio IDE with both conflict notification and communication. Syde [25] uses a fine-grained change tracking mechanism in which object-oriented systems are modeled as abstract syntax trees, and changes are tree operations. To reduce false positives, Crystal [26] merges local repositories in

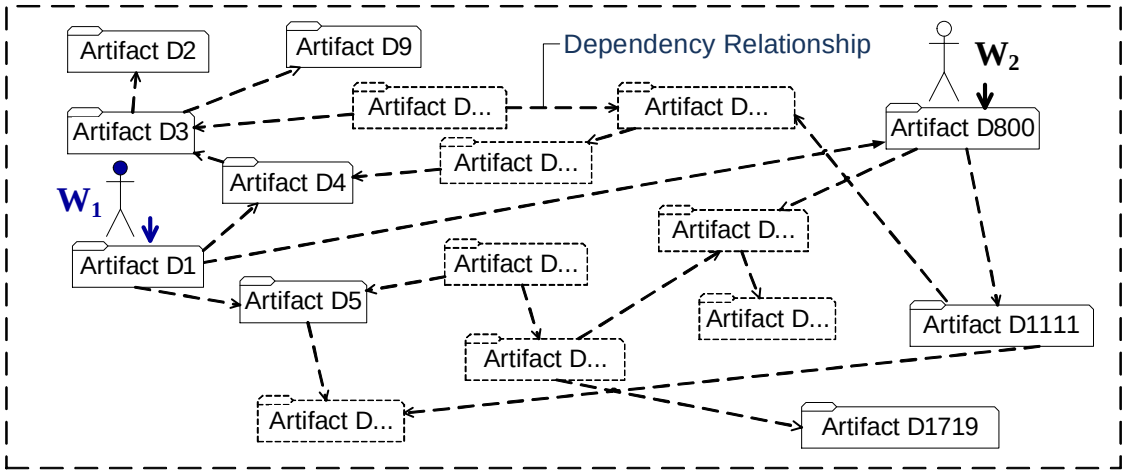


Figure 2.1: Collaborative software development with a large amount of artifacts and the complex dependencies among them

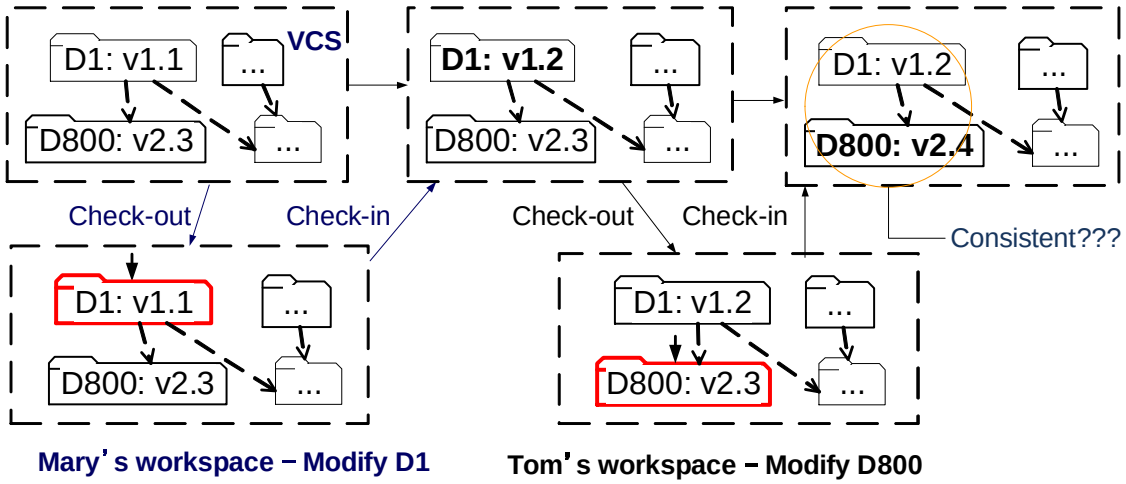


Figure 2.2: Are changes not executed concurrently always safe?

advance to detect pending conflicts rather than potential conflicts. Similarly to Crystal, WeCode [27] computes, rather than predicts, the presence of merge conflicts, but by continuously integrating the changes in the workspaces of workers into a merge workspace shared among the workers.

1. The previous works notified the workers of the concurrent changes that may cause a conflict. However, are changes not executed concurrently always safe?

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 other 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 (Fig. 2.1). Fig. 2.2 shows a simplified situation to illustrate this problem. Mary and Tom change the artifacts $D1$ and $D800$, respectively, at different times. First, Mary checks-out a working copy of the remote repository to her workspace, modifies $D1$, and checks-in her

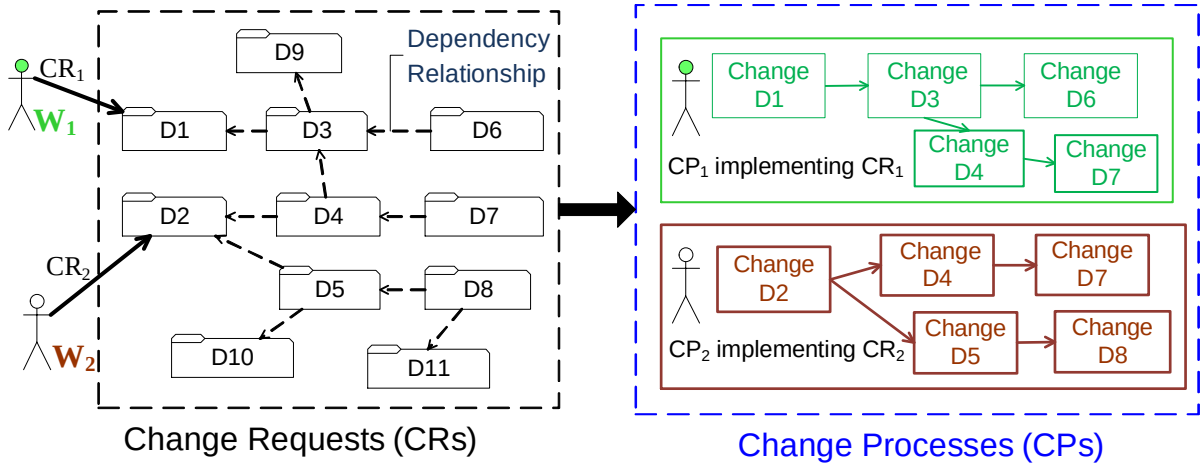


Figure 2.3: Are changes really separated?

changes successfully. Later, Tom wants to change $D800$. We assume that he has changed many artifacts but not $D1$ since the last commit. When he synchronizes his workspace with the repository before starting modifying $D800$, VCS tells him that his merge on $D1$ is clean. As a result, he just focuses on other merge conflicts, and may overlook Mary's change of $D1$. However, $D1$ depends on $D800$. Modifying $D800$ without considering the change history of $D1$ may lead to an inconsistency between the newly generated version $v2.4$ of $D800$ and the version $v1.2$ of $D1$. Therefore, the change history of an artifact and its dependency-related artifacts should be taken into consideration.

2. In the previous works, changes to artifacts are considered individually. However, are changes really separated?

Because of the dependencies among software artifacts, a change to an artifact, called *root artifact*, may affect many other artifacts. A change initiated from a root artifact can spread to other artifacts, which in turn can reach the root artifact by dependency relationships. This means that, a worker often needs a *change process* that is a sequence of tasks applying the changes to many artifacts, to fulfill a change request. Although the change processes are not shown explicitly in the software development environment, they exist and are gradually constructed by the workers. In the example shown in Fig. 2.3, to implement the change request CR_1 with the root artifact $D1$, the worker W_1 may need to implement a change process CP_1 including the changes to the artifacts $D1$, $D3$, $D4$, $D6$, and $D7$. Similarly, from the change request CR_2 on the artifact $D2$, the worker W_2 may have to implement a change process CP_2 including the changes to the artifacts $D2$, $D4$, $D5$, $D7$, and $D8$. Therefore, the changes in a change process are not separated but related to each other through their shared target of realizing the change request, and the dependencies among the artifacts affected by the changes.

Because a change process can reveal much information about a change inside it, for example, the preceding changes, the concurrent changes, and the succeeding changes, the change process of a change can be considered as the *context of the change*. 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. Following is our definition of change process.

Definition 2.1.1. ***(Change Process)*** *A change process (CP) is a sequence of tasks that*

apply changes to a set of artifacts to fulfill a change request.

3. The previous works just provided information about the current changes causing a (potential) conflict. However, is the information about these changes only enough for resolving an inconsistency?

To resolve an inconsistency, the workers need to consider the contexts of the changes causing the inconsistency, including the change history, the ongoing changes, and the planned changes, 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.

Therefore, it is necessary to have a more effective approach to deal with the inconsistency problem in collaborative software developments with regard to the above problems.

2.2 Illustrating Example

Mary and Tom are the developers of a hypothetical airline ticket-sales software system. Fig. 2.4 shows an excerpt of the system in which *VIPCustomer* and *RegularCustomer* are two classes implementing the *Customer* interface. *Display* is a class showing the display screen of customers.

To implement a change request that shows the accumulated point of a customer, the signature *showPoint()* is added to the *Customer* interface (Scenario A). Two empty *showPoint()* methods are added to the *VIPCustomer* and *RegularCustomer* classes at the beginning to avoid compilation errors. In Fig. 2.4, these methods are visible when their implementations are already finished. Next, Mary implements the *showPoint()* methods for the *VIPCustomer* and *RegularCustomer* classes.

First, Mary implements the *showPoint()* method of the *VIPCustomer* class. She synchronizes her workspace before doing her changes to ensure that her workspace is updated with the remote repository. Mary uses the *showCustomerScreen()* method of a third-party class, the *Display* class, in the implementation of the *showPoint()* method (Scenario B). In the morning, she finishes modifying the *VIPCustomer* class. Because she has an urgent meeting with a customer, she checks-in her changes, leaves the office, and delays the implementation of the *showPoint()* method for the *RegularCustomer* class until that night at home.

In the afternoon, the author of the *Display* class, Tom, decides 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 (Scenario C). After finishing modifying the *Display* class, Tom checks-in his update successfully.

At night, Mary checks-out the project to her workspace, and also uses the *showCustomerScreen()* method of the *Display* class in the implementation of the *showPoint()* method for the *RegularCustomer* class. After she finishes updating the *RegularCustomer* class, Mary also checks-in her changes successfully (Scenario D).

Because there are no syntax conflicts on the changed artifacts and these three changes happen sequentially, VCSs and the conflict 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 Tom on the *showCustomerScreen()* method affects the earlier change of Mary on the *showPoint()* method of the

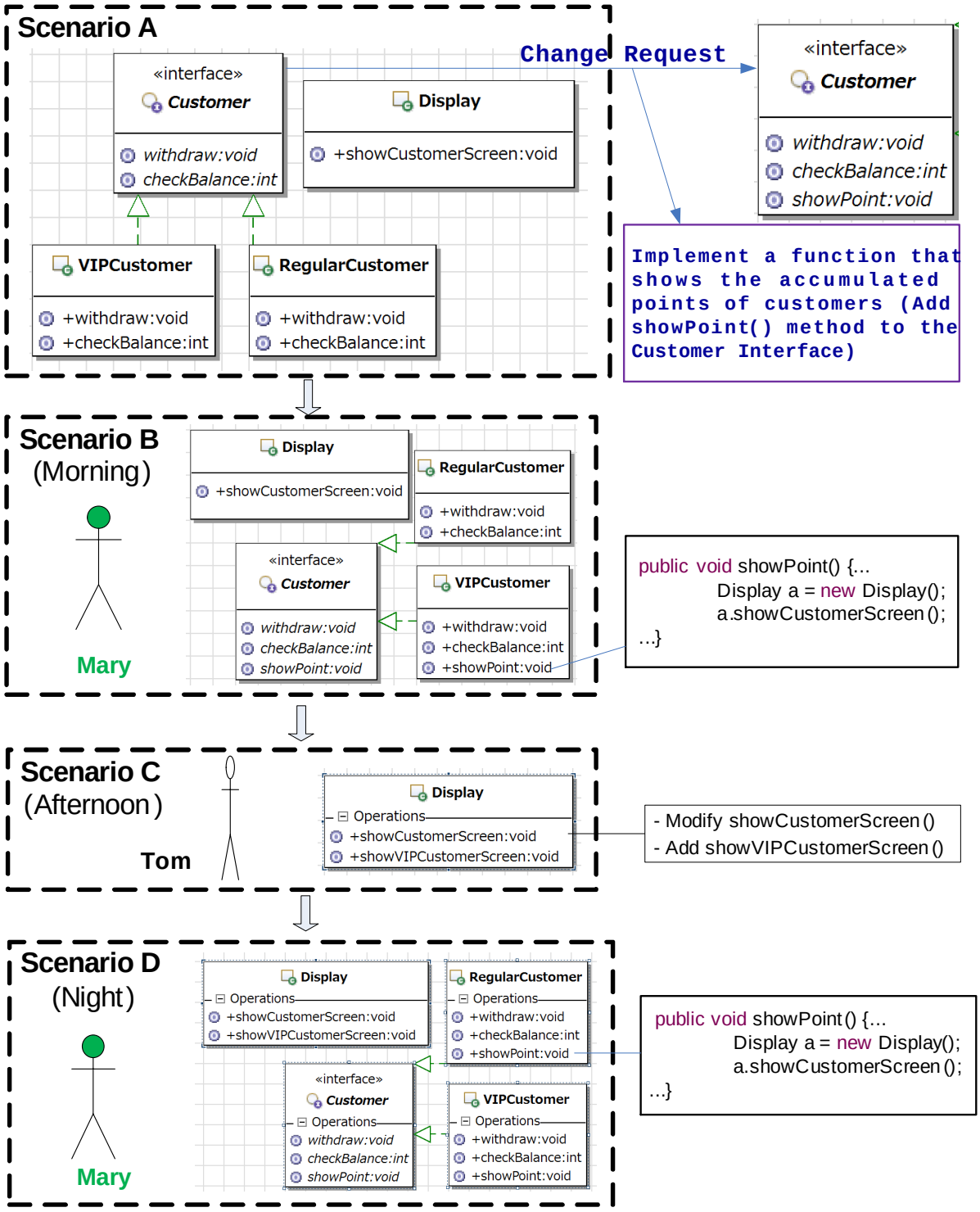


Figure 2.4: Motivating example

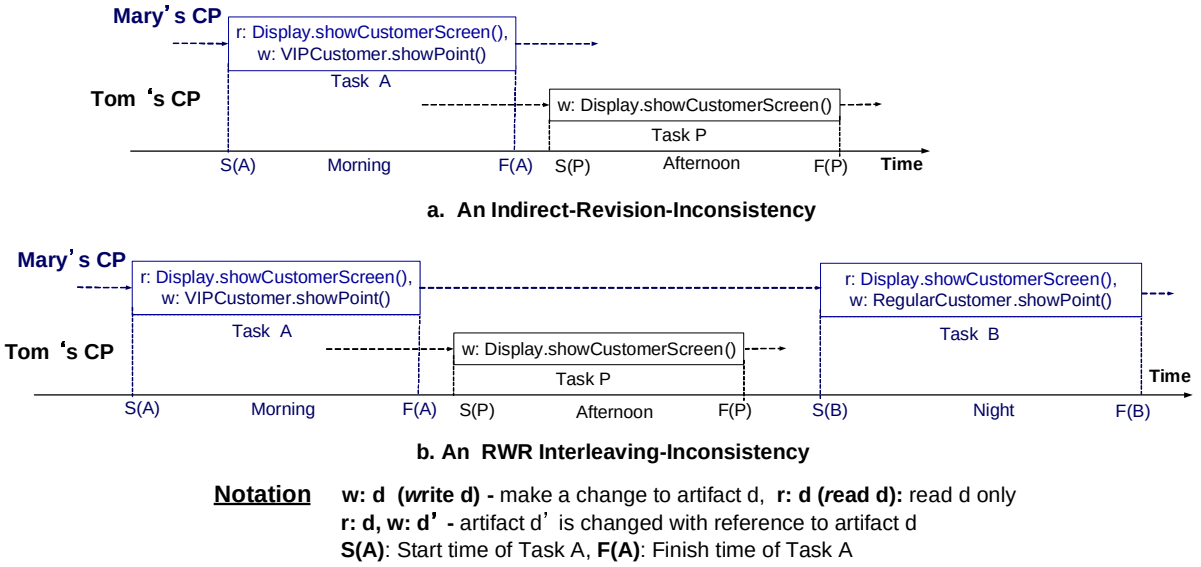


Figure 2.5: The inconsistencies may happen in the motivating example

VIPCustomer class, because Mary implements this method with reference to the *showCustomerScreen()* method. Another inconsistency is **RWR Interleaving-Inconsistency**, in which Mary implements the *showPoint()* methods for the *VIPCustomer* and *RegularCustomer* classes with the same view of the *showCustomerScreen()* method. However, she does not recognize that Tom has modified this method in the interval between her two changes. More explanations of these inconsistencies will be given in Section 3.2.

These inconsistencies could have been avoided or detected before it propagated further, if:

- Tom had notified Mary of his change. However, a worker does not always have sufficient information about the work of other workers to notify them of the change that can affect their works. Performing all the required tasks, for instance, finding the works impacted by his change, specifying the change, and sending to the impacted workers, by himself is tricky and time-consuming. Also, if the VCSs tell a worker that a merge is clean, he will often not suspect the new changes.

- Mary had recognized the changes of Tom and the impact of his changes on her work, and had revised the implementation of the *showPoint()* method in the *VIPCustomer* and *RegularCustomer* classes, with regard to the updated version of the *showCustomerScreen()* method and the new method, *showVIPCustomerScreen()*, of the *Display* class. Nonetheless, a worker does not always pay attention to the changes of other workers, except for the changes causing merge conflicts, when she synchronizes her workspace with the remote repository. In addition, it is difficult for her to remember in detail the contents of all the changes she has made before. It will be more difficult to recognize this situation, if the task of updating the *VIPCustomer* class is assigned to a different worker, for example Peter.

- Mary had locked all the involved artifacts. However, because of her long-term change process, locking the artifacts in use will affect the performance of other workers in

her team. In the database area, DBMS provides some concurrency control techniques to ensure the noninterference or isolation property of concurrently executing transactions. Nevertheless, the database transactions also require some critical data to be locked during the transactions which take only a few seconds. Therefore, applying the concurrency control techniques of DBMS to this situation is unsuitable because of the long-running nature of the change processes in collaborative software development environment.

- Mary and Tom had been notified of the previous changes of other workers that may affect their current work, along with the context of the changes, to help them understand the situation easily and quickly.

Based on these considerations, if there is a system that supports workers to manage their change processes, and provides the workers with the contexts of the changes that may affect their work or be affected by their work, inconsistencies, including conflicts, can be prevented, detected and resolved more effectively.

2.3 Approach Overview

VCSs can detect a direct conflict when a worker commits his changes or synchronizes his workspace with the remote repository. The worker who checks-in later must resolve this conflict by himself, or by negotiating with the previously check-in worker. In the case of indirect conflict, VCSs cannot help, because the changes are implemented on different artifacts. Recent studies [21, 24, 18, 20, 25, 28, 26] can detect emerging indirect and direct conflicts, by monitoring the ongoing changes in the workspaces of the workers and notifying them when there are concurrent changes to the same artifact or to dependency-related artifacts.

The inconsistencies related to non-concurrent changes and the contexts of changes are not mentioned in these studies. However, changing an existing artifact to fix an error or to implement a new change request is done regularly during the development of a software system. Changing an artifact without understanding its change history and the purpose of the past changes on it can introduce other types of inconsistency. Therefore, we need to control the changes to an artifact by managing who have made the changes to the artifact, what changes they have made, and the purpose of their changes. Because a change process can supply much information about the changes inside it, such as the change purpose, current changes, past changes, and future changes, managing the change processes helps the related workers have a global view of what is happening in the system to cooperate with others in preventing, detecting, and resolving inconsistencies more effectively.

In summary, our approach is motivated by the following points.

- Besides conflicts mentioned in the previous studies, there are other inconsistencies involving non-concurrent changes and the contexts of changes, and they are detected much later during the build process, integration test, or even runtime failure.

- A change request is often implemented by a change process that includes many changes applied to the artifacts connected by dependencies, and the changes in a change process are often implemented in a consistent manner.

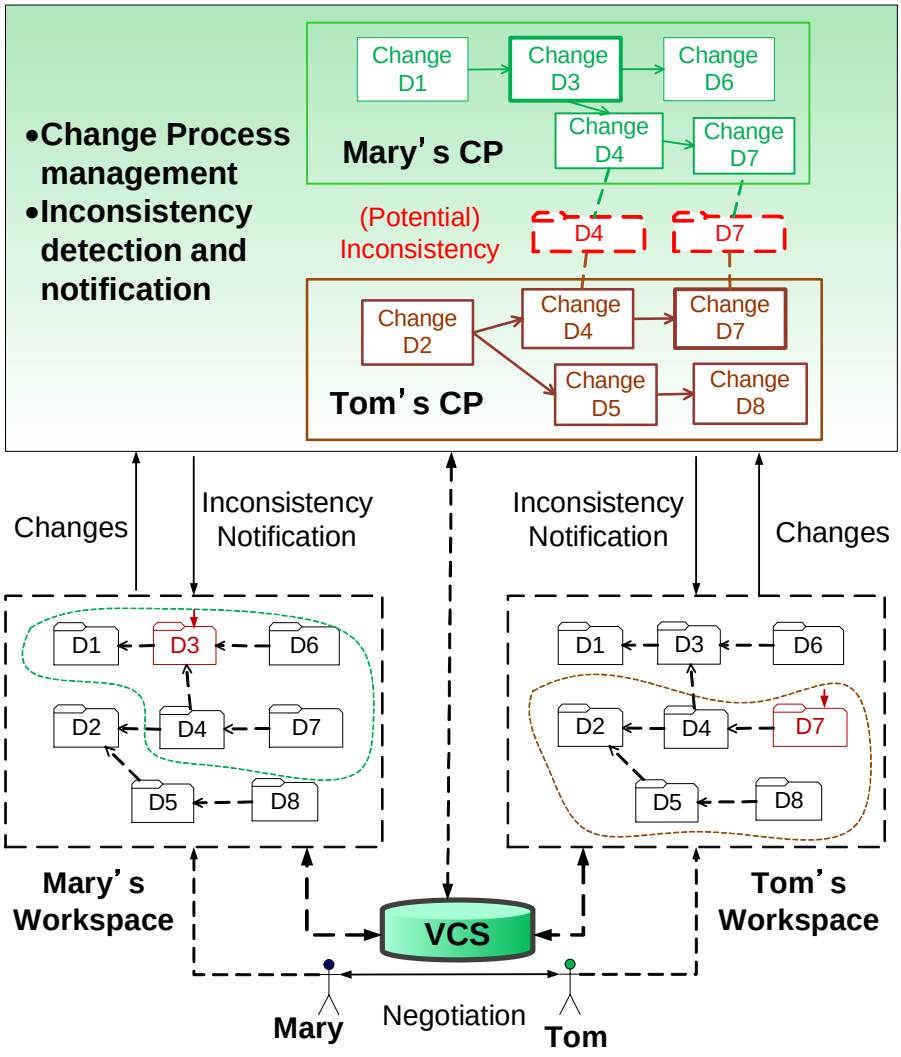


Figure 2.6: Approach to handling inconsistencies

- To resolve an inconsistency, a worker needs to consider not only his changes in the past, present, and future, but also the changes made by the workers involved.
- A worker can not always be aware of the changes or the impact of the changes made by other workers, to his work because of the large number of software artifacts and the complex dependencies among them. Also, it is difficult for a worker to remember the contents of all the changes he has made and to guarantee the consistencies of effects by his changes, specially if he works on multiple projects concurrently.
- Like other planning types, setting a change plan to fulfill a change request gives the workers a path to follow and helps them understand their work more clearly. A change plan can also be a good communication tool for the workers to understand the goal and activities of each other.

Based on the above considerations, we propose an approach that deals with inconsistency in collaborative software development more effectively as follows.

- Besides handling the conflicts like the previous works, we pay attention to other

types of inconsistencies by considering the non-concurrent changes and the contexts of the changes that are the change processes containing the changes. We define the patterns of inconsistency with regard to the relationships between the affected artifacts, same artifact or different artifacts, the time orders of the tasks applying the changes to the artifacts, concurrent or not, and the contexts of the changes. Each pattern of inconsistency represents an inconsistency problem in collaborative software development. Defining the patterns of inconsistency 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.

- Our approach is to represent the change processes in a collaborative software development environment explicitly, manage their execution, and share information about the change processes among the workers to handle inconsistencies more effectively. Fig. 2.6 shows an overview of our approach. To detect and resolve inconsistencies more effectively, we combine the *workspace awareness* technique and the *context awareness* technique [29]. Context awareness means sharing information about the context of a change that is the change process containing the change. By monitoring the progress of the change processes and the ongoing changes in the client workspaces, we can obtain the latest information about the changes in the system for inconsistency analysis, and 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. The context of an inconsistency will help the workers understand and resolve the inconsistency more easily compared with the previous works that supply the changes causing the inconsistency only.

Detail of the patterns of inconsistency and our approach will be presented in the next chapters.

2.4 Summary

Our research is aimed at managing inconsistencies in collaborative software development more effectively. Our approach is motivated by the problems that have not been addressed by the existing studies in this area. Neglecting the relationships among the changes implemented to fulfill a change request, focusing only on the ongoing changes that are concurrent, and revealing little information about the contexts of the detected inconsistencies by these studies motivate us to deal with the inconsistency problem at the level of the contexts of changes, which are the change processes containing the changes, rather than just the changes themselves.

Chapter 3

Patterns of Inconsistency

In this chapter, we describes the patterns of inconsistency that we define and classify by considering the change processes containing the changes and the non-concurrent changes in addition to the relationships among the changed artifacts and the concurrent changes like the related studies.

3.1 Overview

An inconsistency pattern 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 in collaborative software development environments more easily. The patterns are themselves easier to be improved. When we encounter an inconsistency that has not yet been described, we can create a new pattern for dealing with this inconsistency, and add it to the collection of the patterns of inconsistency. In addition, although solution to a pattern of inconsistency varies from situation to situation, it should be described as a property of the pattern for reference in the future. Similarly, when a problem arises with a specific pattern, we can track it back to the pattern, and revise it accordingly to cover the problem better. Following is our formal definition of inconsistency.

Definition 3.1.1. ***(Inconsistency)*** *Inconsistency is 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. This situation leads to syntactic errors or semantic errors causing unexpected or unintended behaviors of the constructed software system.*

The previous works in inconsistency awareness concentrated on the concurrent changes and the relationships among the affected artifacts. They identified two patterns of inconsistency: direct conflict and indirect conflict. Direct conflicts are caused by concurrent changes to the same artifact. Indirect conflicts are caused by changes to one artifact affecting concurrent changes to another artifact. In this dissertation, besides handling the conflicts like the previous works, we also pay attention to other types of inconsistency, by considering not only the relationships among the affected artifacts and the concurrent changes but also the non-concurrent changes and the contexts of the changes that are the change processes containing the changes. In other words, we identify the patterns

of inconsistency with regard to three properties: the relationships between the affected artifacts, same or not, the time orders of the tasks applying the changes to the artifacts, concurrent or not, and the contexts of the changes, the changes in the same change process or in different change processes.

- It is easy to understand why the relationships among the artifacts are the foremost property that we must consider, because the artifacts are directly impacted by the changes, and a change can affect other changes only if there are some relationships among the artifacts they apply to. If the tasks apply changes to the same artifact, the mutual influences among them are clear. However, if the tasks applying changes to different artifacts, the mutual influences among them depend on the content of the changes and the type of the dependencies among the affected artifacts. Therefore, we must consider two cases: same artifact or different artifacts.

- Regarding the concurrency of the changes, paying attention to concurrent changes like the previous studies is necessary but not enough. As we have discussed in Section 2.1, inconsistencies may still happen even when the changes are not concurrent, if a worker does not recognize the impact of the previous changes, made before by other workers, on his ongoing changes, or the impact of his ongoing changes on the previous changes. This problem becomes common along with the increase in the number of workers, software scale, and development duration. If the tasks are happening concurrently, the changes they made have not been committed yet. In other words, the effects of these changes to the constructed software system are temporarily happening in their local workspaces only. In the case of non-concurrent tasks, the previous tasks have committed their changes to the remote repository, and everyone can access the updated version. Hence, we have distinguished the concurrent tasks from non-concurrent tasks.

- The third property relates to the contexts of the changes that are the change processes containing the changes. The changes in a change process are not separated but related to each other, and are often implemented in a consistent manner toward the share target of realizing the assigned change request (See Section 2.1). Changes in different change processes will have weaker connection, compared with the changes in the same change process. Therefore, we should take into account the context of the changes, i.e. the changes in the same change process or in different change processes.

Although an inconsistency may involve many changes, we can decompose this multi-relationship (multi-tier relationship) into many pairwise-relationships (two-tier relationships). Therefore, to identify the patterns of inconsistency, we examine the situations that can lead to inconsistencies between two changes first. Then, the patterns of inconsistency are identified by generalizing these basic situations. As a worker can understand earlier changes made before by other workers in the same change process shared among them, we assume that later changes are made considering their impacts on the previous changes to ensure their consistency. This means that in the same change process, even if the later changes to an artifact contradict the earlier changes to the artifact, they are still under control and no need to worry about them. This dissertation focuses more on unexpected influences caused by a task in a change process on tasks in different change processes.

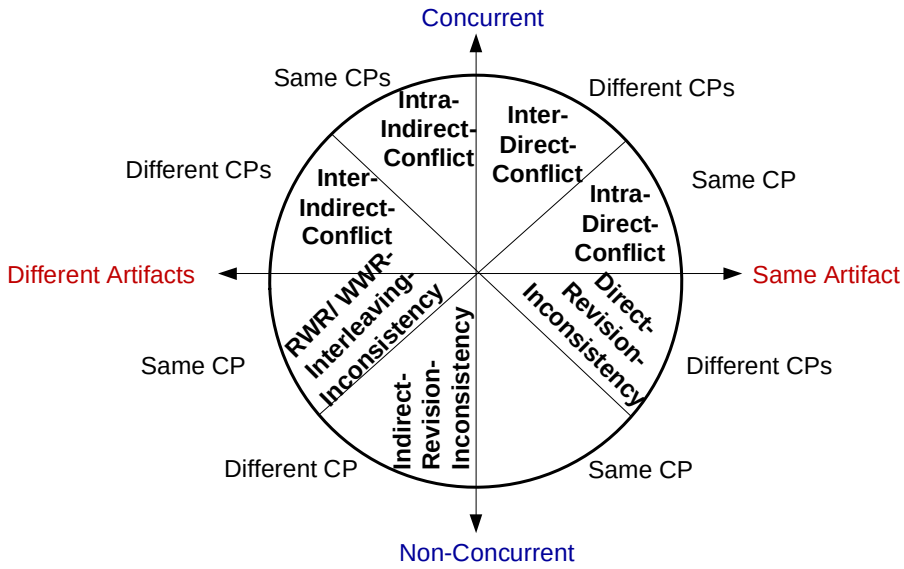


Figure 3.1: Categories of inconsistency

Currently, we have identified eight patterns categorized based on the three mentioned properties, as shown in Fig. 3.1.

- **Intra-Direct-Conflict** is an inconsistency in which concurrent tasks in the same change process change the same artifact.
- **Inter-Direct-Conflict** is an inconsistency in which concurrent tasks in different change processes change the same artifact.
- **Intra-Indirect-Conflict** is an inconsistency in which a change to an artifact affects the concurrent changes to other artifacts by tasks in the same change process.
- **Inter-Indirect-Conflict** is an inconsistency in which a change to an artifact affects the concurrent changes to other artifacts by tasks in different change processes.
- **Direct-Revision-Inconsistency** is an inconsistency in which there are contradictory intentions in revising the same artifact at different times by tasks in different change processes.
- **Indirect-Revision-Inconsistency** is an inconsistency in which a later change to an artifact affects earlier changes to other artifacts by tasks in different change processes.
- **Interleaving-Inconsistency** is an inconsistency in which there are inconsistent views of using a shared artifact by two tasks in the same change process, because the shared artifact is modified by a task in another change process at sometime during the interval between these two tasks.

In Fig. 3.1, the combination relating to non-concurrent tasks applying changes to the same artifact in the same change process is left empty because of our above assumption about the consistency among non-concurrent tasks in the same change process.

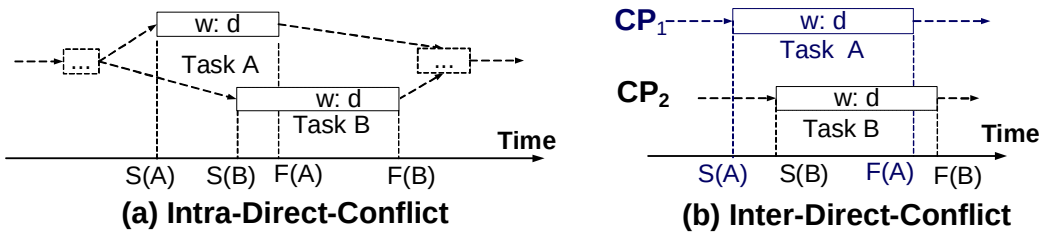


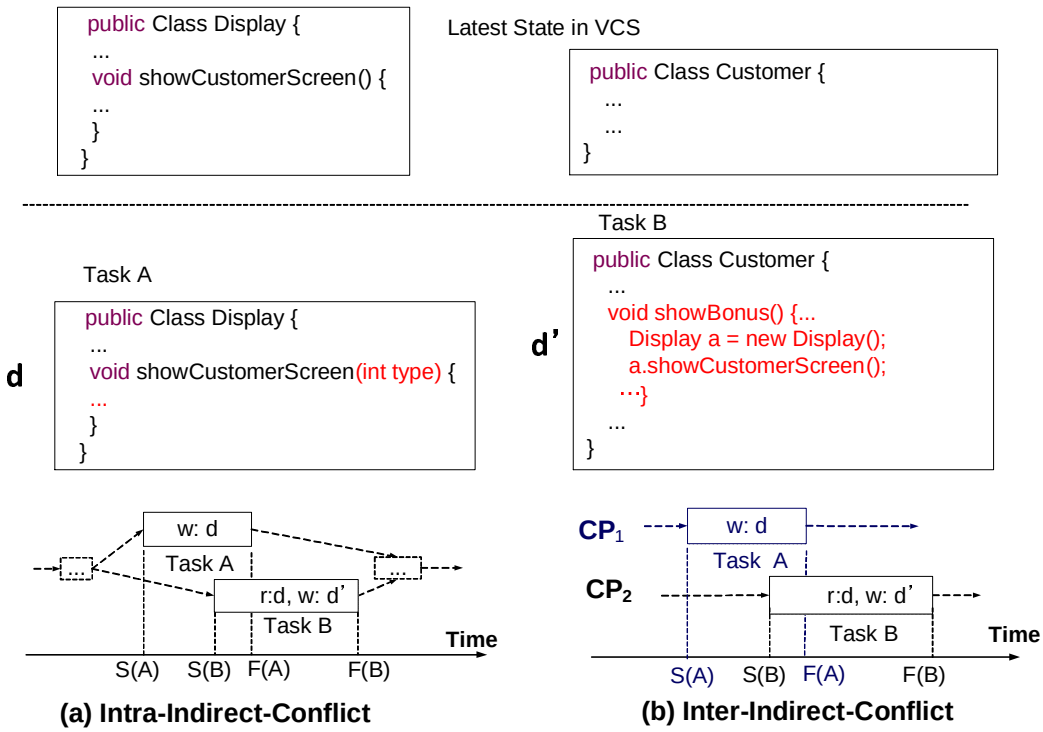
Figure 3.2: Direct-Conflict patterns

We expect that our patterns of inconsistency are completed in the scope of the three properties that we use to classify them. However, we can identify more specific cases of each type of inconsistency. For example, we have identified the RWR Interleaving-Inconsistency and WWR Interleaving-Inconsistency patterns as two specific cases of Interleaving-Inconsistency. Exploring other specific cases of each type of inconsistency is our future work.

3.2 Patterns of Inconsistency

We assume that each task applying changes to artifacts belongs to a change process. For each task in a change process, the worker will check-out the latest versions of the artifacts in the remote repository to his workspace when he starts the task, and will check-in the changed artifacts when he finishes the task. The following notations are used in our definitions of the patterns of inconsistency:

- **w: d** Make a change (*write*) to Artifact d
- **r: d** Artifact d is *read*-only
- **r: d, w: d'** Artifact d' is changed with reference to d
- **S(A), F(A)** Start time and Finish time of Task A
- **[S(A),F(A)]** Execution Time of Task A
- **d' - - > d** Artifact d' depends on Artifact d (dependency relationship)
- **CO(A,d)** Version of Artifact d when it is checked-out at the beginning of Task A
- **CI(A,d)** Updated version of Artifact d when it is checked-in at the end of Task A



Notation **w: d (*write* d)** Make a change to Artifact d, **r: d (*read* d)** Read d only
r: d, w: d' Artifact d' is changed with reference to Artifact d
S(A): Start time of Task A, **F(A)**: Finish time of Task A

Figure 3.3: Example of Indirect-Conflict patterns

3.2.1 Intra-Direct-Conflict

A situation in which the tasks in the same change process concurrently change (***write***) the same version of an artifact, and create new conflicting versions.

Fig. 3.2a describes an Intra-Direct-Conflict relating to two concurrent tasks, A and B, in the same change process:

- A and B are concurrent: $[S(A), F(A)] \cap [S(B), F(B)] \neq \emptyset$.
- Check-in conflict: $CO(A, d) = CO(B, d)$ and $CI(A, d) \neq CI(B, d)$.

3.2.2 Inter-Direct-Conflict

A situation in which the tasks in different change processes concurrently change (***write***) the same version of an artifact, and create new conflicting versions.

Fig. 3.2b describes an Inter-Direct-Conflict relating two concurrent tasks, A and B, in different change processes:

- A and B are concurrent: $[S(A), F(A)] \cap [S(B), F(B)] \neq \emptyset$.
- Check-in conflict: $CO(A, d) = CO(B, d)$ and $CI(A, d) \neq CI(B, d)$.

3.2.3 Intra-Indirect-Conflict

A situation in which an artifact d' is assigned a value different from the intention of a worker, because he does not recognize that a concurrent task, made by another worker in the same change process, is applying changes to the same version of an artifact d that he is using (***r**eading*) to change (***w**rite*) d'.

Fig. 3.3a describes an example of an Inter-Indirect-Conflict in which:

- Task A and Task B are concurrent tasks in the same change process.
- Task A is modifying the signature and the content of the $showCustomerScreen()$ method in the $Display$ class, while Task B is adding a new method $showBonus()$, which invokes the $showCustomerScreen()$ method, into the $Customer$ class.

3.2.4 Inter-Indirect-Conflict

A situation in which an artifact d' is assigned a value different from the intention of a worker, because he does not recognize a concurrent change, made by a task in a different change process, is applying changes to the same version of an artifact d that he is using (***r**eading*) to change (***w**rite*) d'.

Fig. 3.3b describes an example of an Inter-Indirect-Conflict in which:

- Task A and Task B are concurrent tasks in different change processes.
- Task A is modifying the signature and the content of the $showCustomerScreen()$ method in the $Display$ class, while Task B is adding a new method $showBonus()$, which invokes the $showCustomerScreen()$ method, into the $Customer$ class.

3.2.5 Direct-Revision-Inconsistency

A situation in which an artifact is assigned an unexpected or unintended value because some later workers, who do not understand correctly the purpose of the previous changes made by other workers, make contradictory changes (***w**riting*) with the previous changes while executing their tasks in different change processes.

Please note that this pattern does not imply revisions that are performed carefully considering the impact of their changes on the previous works, to fix a bug or add some new features during the software development and evolution.

Also, Direct-Revision-Inconsistency can be considered as the cause of the Indirect-Revision-Inconsistency and Interleaving-Inconsistency patterns, in the case the workers do not pay attention to the changes between two revisions, or do not understand correctly the purpose of the previous changes.

Fig. 3.4 describes an example of a Direct-Revision-Inconsistency in which:

- Task A and Task B are non-concurrent tasks in different change processes. A happens before B, $F(A) < S(B)$.
- The worker performing Task A added a $fly()$ method to the $Bird$ class. She used a field $canFly$ to denote the ability to fly of birds. In the constructor of $Bird$, she set the value of $canFly$ to $false$, because she thought that this helps specify the flying ability of the bird species more conveniently. With flying birds, for example Eagle,

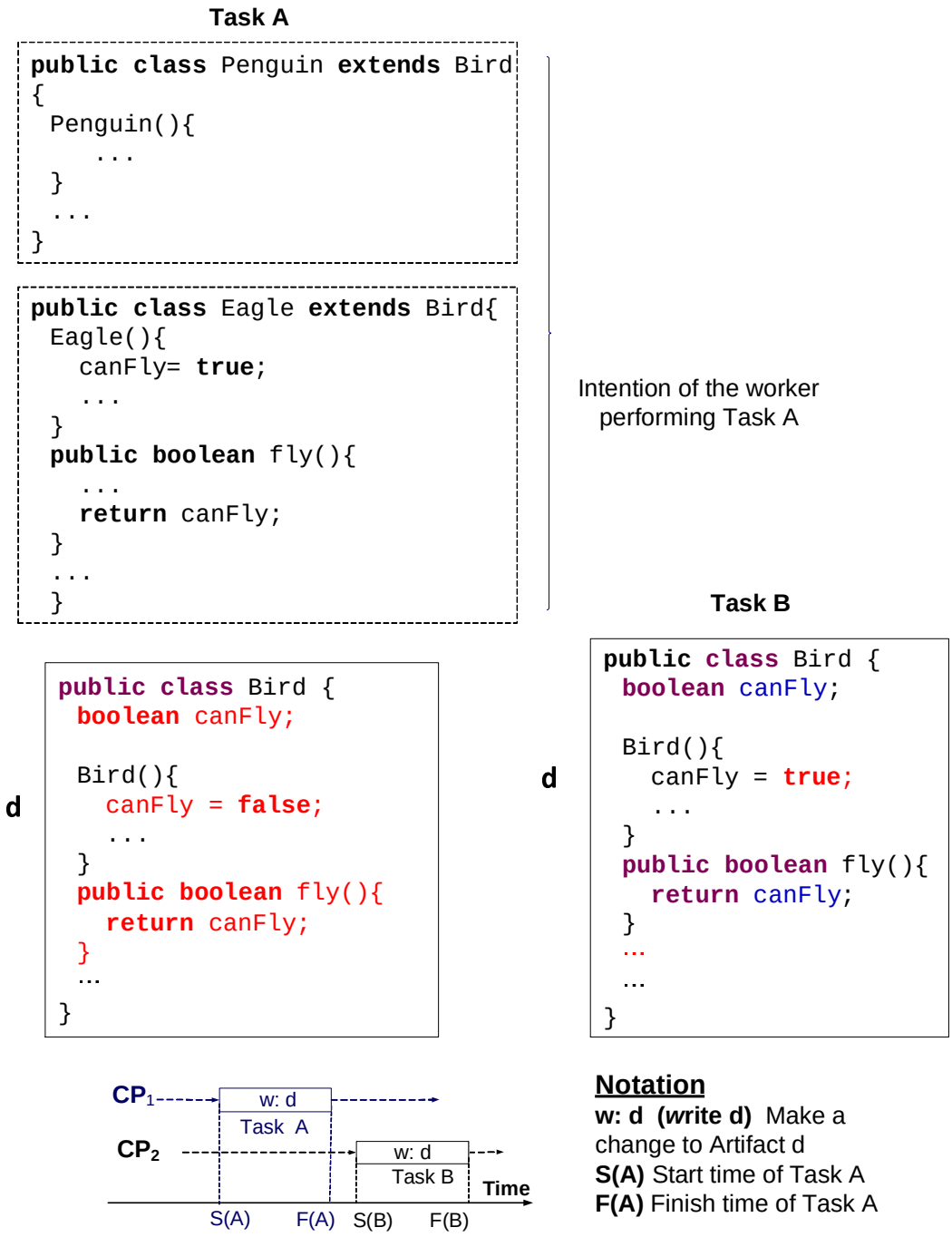


Figure 3.4: An example of Direct-Revision-Inconsistency pattern

one should change the value of *canFly* to *true* in the constructor, and override the *fly*() method to provide more information about this ability. With flightless birds, for example Penguin, this function can be ignored.

- Later, another worker performing Task *B* changed the default value of the field *canFly* to *true*, because he thought that most birds can fly.

In this example, the later worker did contradictory changes with the previous changes without carefully considering the purpose and the work of the previous worker.

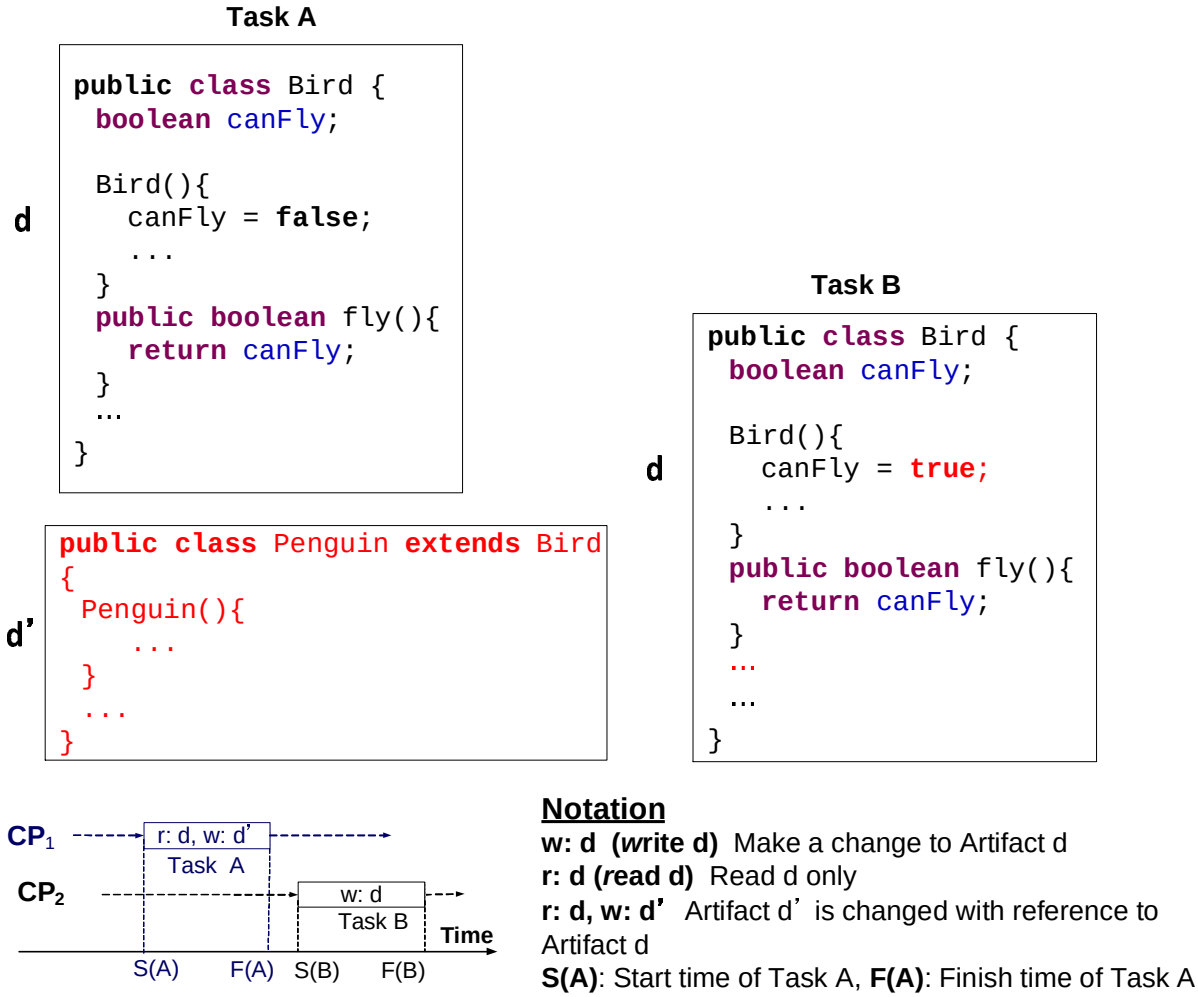


Figure 3.5: An example of Indirect-Revision-Inconsistency pattern

3.2.6 Indirect-Revision-Inconsistency

A situation in which an artifact d' is assigned a value different from the intention of a worker who made the latest changes (*writing*) to d', because some later workers change (*write*) an artifact d, which he used (*read*) to change (*write*) d', without considering the effect of their changes to d'.

Fig. 3.5 describes a simple example of an Indirect-Revision-Inconsistency in which:

- Task A and Task B are non-concurrent tasks in different change processes. A happens before B, $F(A) < S(B)$.
- The worker performing Task A implemented a new class $Penguin$ extending the class $Bird$. Because Penguin can not fly and the default value of the field $canFly$ in the superclass $Bird$ is $false$, changing the value of this field in the constructor of $Penguin$ and overriding the $fly()$ method are not necessary.
- Later, another worker executing Task B modified the default value of the field $canFly$ to $true$, because he thought that most birds can fly.

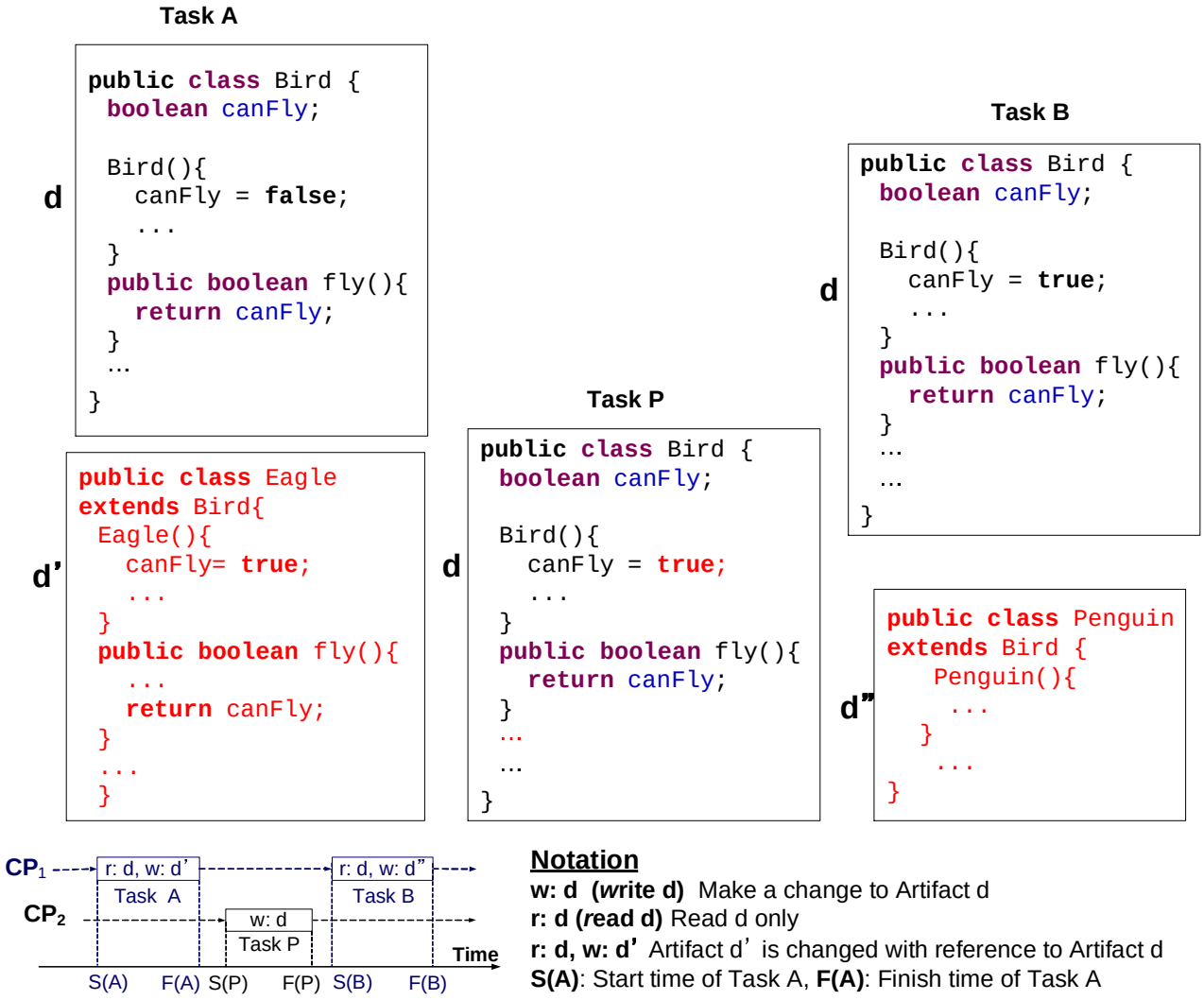


Figure 3.6: An example of RWR Interleaving-Inconsistency pattern

In this example, the later worker did not recognize the impact of his change on the behavior of $Penguin$. As a result, $Penguin.fly()$ will unexpectedly return $true$ that should be $false$.

3.2.7 RWR Interleaving-Inconsistency

A situation in which some artifacts are assigned values different from the intention of a worker who performs two tasks A and B in the same change process to change (***write***) these artifacts, because he does not recognize that a task in a different change process changed (***wrote***) an artifact d, that is used (***read***) by both A and B to change (***write***) these artifacts, at sometime during the interval between A and B.

Fig. 3.6 describes an example of an RWR Interleaving-Inconsistency in which:

- Task A and Task B are in the same change process, CP_1, that is different from the change process of Task P, CP_2. P happens after A and before B, $[S(P), F(P)] \subset [F(A), S(B)]$.

- The first worker, performing Task A and Task B, implemented the $Eagle$ class inheriting the $Bird$ class first. Examining the $Bird$ class, she recognized that the $fly()$ method always returns the value of the $canFly$ field that is set to $false$ by default. Therefore, when implementing the $Eagle$ class, which represents Eagle, a flying bird, she changed the value of $canFly$ to $true$ in the constructor, and overrode the $fly()$ method to provide more information about this ability. After finishing the $Eagle$ class, she intended to implement the $Penguin$ class. With flightless birds like Penguin, this function can be ignored. However, because she had an urgent meeting, she delayed implementing the $Penguin$ class later, checked-in her changes, and left the office.

- When the first worker was away, another worker performing Task P modified the default value of the $canFly$ field in the $Bird$ class to $true$, because he thought that most birds can fly.

- Later, when the first worker came back from the meeting, she performed Task B that implemented the $Penguin$ class with her original intention, without considering the new changes on the $Bird$ class. Because Penguin is a flightless bird, she ignored implementing and testing functions related to the flying ability.

In this example, the first worker did not recognize the interleaving changes made by the second worker to the $Bird$ class. As a result, $Penguin.fly()$ will unexpectedly return $true$ that should be $false$. If she had recognized the changes of the $Bird$ class, she could have set the value of $canFly$ to $false$ in the constructor of the $Penguin$ class. Also, she might have implemented the $Eagle$ class in a different way, for example, she would have removed the statement that sets the value of $canFly$ to $true$ from the constructor of $Eagle$.

3.2.8 WWR Interleaving-Inconsistency

A situation in which some artifacts are assigned values different from the intention of a worker who performs two tasks A and B in the same change process to change (***w**rite*) these artifacts, because he does not recognize that another task in a different change process changed (***w**rote*) the value of the artifact d, that was assigned (***w**ritten*) earlier by Task A and was intended to be used (***r**ead*) by Task B, at sometime during the interval between A and B.

Fig. 3.7 describes an example of a WWR Interleaving-Inconsistency in which:

- Task A and Task B are in the same change process that is different from the change process of Task P. P happens after A and before B, $[S(P), F(P)] \subset [F(A), S(B)]$.

- The worker performing Task A added a $fly()$ method to the $Bird$ class. She used a field $canFly$ to denote the ability to fly of birds. In the constructor of $Bird$, she set the default value of $canFly$ to $false$, because she thought that this helps specify the flying ability of the bird species more conveniently. With flying birds, for example Eagle, one should change the value of $canFly$ to $true$ in the constructor, and override the $fly()$ method to provide more information about this ability. With flightless birds, for example Penguin, this function can be ignored. She intended

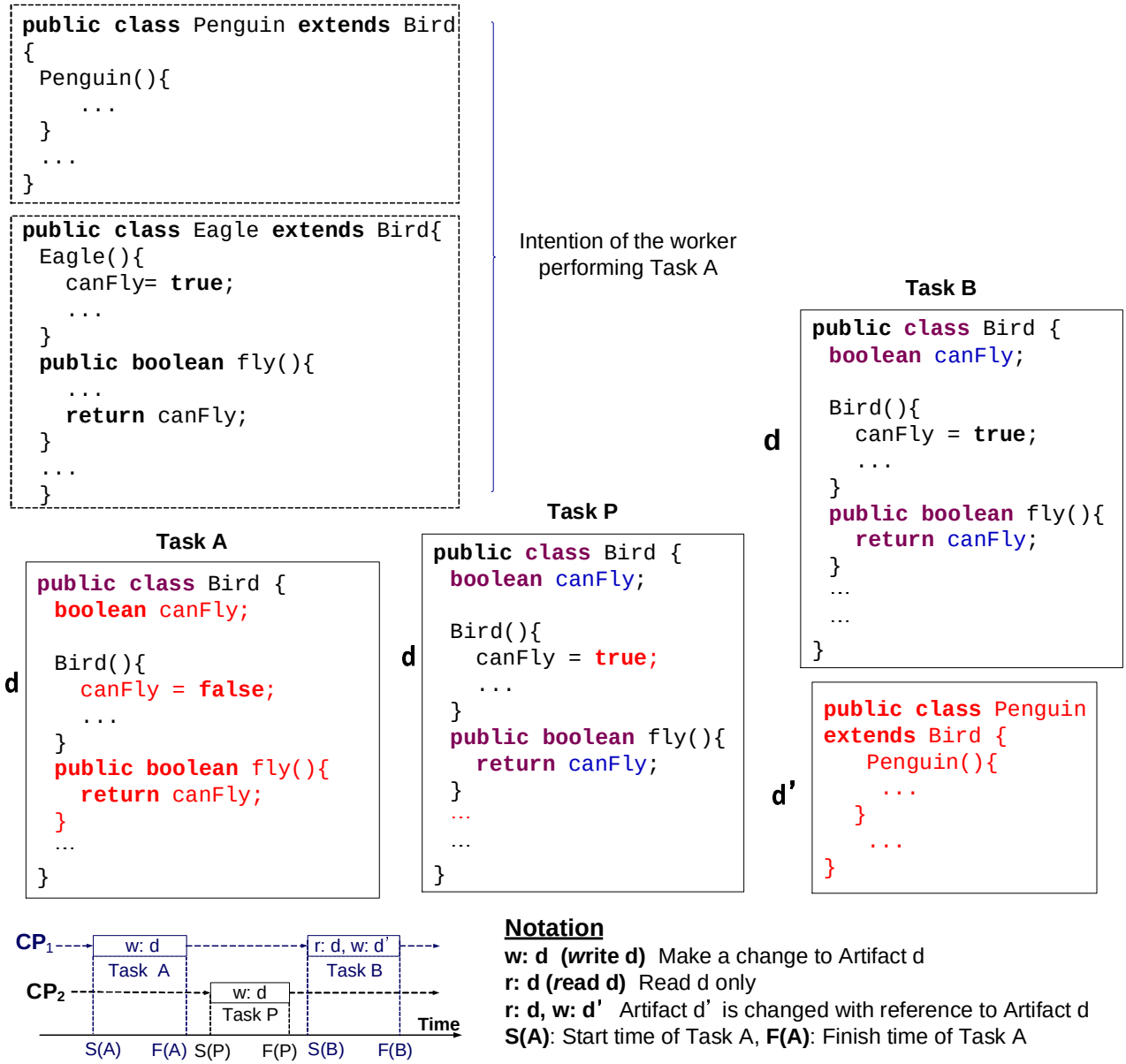


Figure 3.7: An example of WWR Interleaving-Inconsistency pattern

to implement the *Penguin* class, a subclass of *Bird*. However, because she had an urgent meeting, she checked-in her changes and left the office.

- When the first worker was away, another worker performing Task P modified the default value of the *canFly* field in the *Bird* class to *true*, because he thought that most birds can fly.

- Later, when the first worker came back from the meeting, she performed Task B which implemented the *Penguin* class with her original intention, without considering the new changes on the *Bird* class. Also, Penguin is a flightless bird, she skipped testing the functions related to the flying ability.

In this example, the first worker did not recognize the interleaving changes made by

the second worker to the $Bird$ class. As a result, $Penguin.fly()$ will unexpectedly return $true$ that should be $false$. Also, the behavior of the $Bird$ class has been changed different from the intention of the first worker.

3.3 Summary

We have defined the patterns of inconsistency classified based on three properties: the relationships between the changed artifacts, same or different, the time order of tasks applying the changes to the artifacts, concurrent or not, and the context of the changes, in the same or different change processes. These patterns are used as the foundation for recognizing inconsistencies in collaborative software development.

Chapter 4

Theoretical Model of Change Support Environment

In this chapter, we present the theoretical model of Change Support Environment that represents the change processes in the system explicitly as Change Support Workflows and manages their execution based on the patterns of inconsistency.

4.1 Change Support Workflow (CSW)

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 change activities, change orders, artifacts accessed by the change activities, change types, change workers, and the execution times of the change activities. Following is the formal definition of a CSW.

Definition 4.1.1. ***(Change Support Workflow)***
A CSW is a tuple $< id, A, AT, E, D, C, CT, W, GDT, GD, GC, GW, GT >$ *where:*

- *id is the workflow identifier.*
- *A is a set of change activities.*
- $AT = \{$*AND-split, AND-join, OR-Split, OR-join*$\}$ *is a set of types of change activities. An AND-split activity allows its outgoing activities to be performed simultaneously. An AND-join activity can be started only if all its incoming activities have been finished. An OR-split activity allows only one of its outgoing activities to be performed. An OR-join activity can be started if one of its incoming activities has been finished.*
- $E \subseteq (A \times A)$ *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.*
- $CT = \{r, w\}$ *is a set of types of change to artifacts (r:* ***r****ead, w:* ***w****rite).* ***r****ead means that this artifact is for reference only.* ***w****rite means that this artifact is changed.*

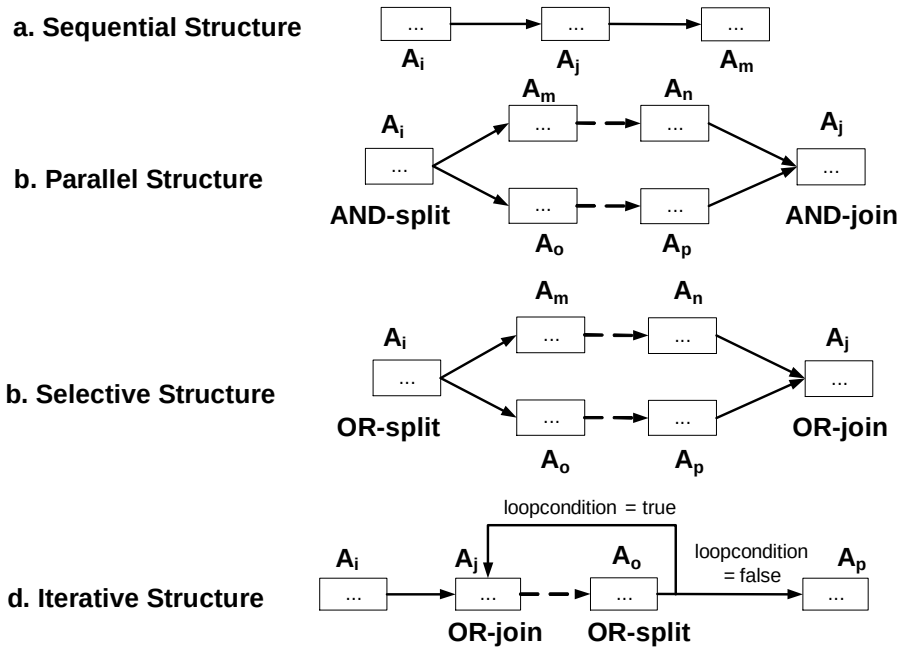


Figure 4.1: Basic control structures of a CSW

- *W is a set of workers who execute the activities of the CSW.*
- *$GDT : C \times CT \rightarrow 2^D$ is a function that returns a set of artifacts associated with a change and a change type.*
- *$GD : C \rightarrow 2^D$ 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**.*

Fig. 4.1 shows the basic control structures of a CSW. In a sequential structure, activities need to be carried out sequentially. Fig. 4.1a shows an example of a sequential structure. Regarding a parallel structure, there is more than one activity that can be carried out at the same time. A parallel structure begins with an AND-split activity and often ends with an AND-join activity. Fig. 4.1b is an example of a parallel structure. As shown in Fig. 4.1c, a selective structure begins with an OR-split activity and often ends with an OR-join activity. In an iterative structure, some activities can be performed more than once. An OR-split activity and OR-join activity are used to control the loop. Fig. 4.1d shows an example of this structure.

4.2 Change Support Environment (CSE)

A Change Support Environment is a collection of CSWs defined or executed within a time interval.

Definition 4.2.1. ***(Change Support Environment)***
A $\mathcal{CSE}$ is a tuple $< \mathcal{CSW}, EST, LFT, \mathcal{CR}, \mathcal{A}, \mathcal{GCSW}, \mathcal{GCR} >$ where:

- $\mathcal{CSW} = \{csw_1, csw_2, ..., csw_n\}$ *is a set of CSWs where* $csw_i =< id_i, A_i, E_i, D_i, C_i, W_i, GD_i, GC_i, GW_i, GT_i >$.
- EST *is the earliest time a CSW is defined or executed.*
- LFT *is the latest time a CSW is defined or executed.*
- $\mathcal{CR}$ *is a set of change requests implemented by the CSWs.*
- $\mathcal{A} = A_1 \cup A_2 \cup ... \cup A_n$ *is a set of change activities of the CSWs. For each activity* a_{ij} *in* $\mathcal{A}$, $GT(a_{ij}) = [S(a_{ij}, F(a_{ij}] \subseteq [EST, LFT]$. a_{ij} *is an activity of a CSW* csw_i.
- $\mathcal{GCSW} : \mathcal{CR} \rightarrow \mathcal{CSW}$ *is a function that returns the CSW implementing a change request.*
- $\mathcal{GCR} : \mathcal{CSW} \rightarrow \mathcal{CR}$ *is a function that returns the change request implemented by a CSW.*

4.3 Inconsistency

In a software development environment, many change requests to a software system, such as adding or modifying a feature, or correcting an error, need to be carried out as soon as possible. Therefore, in a given time interval, there are always many CSWs applying changes to the same software system or to the same part or related parts of a software system, to implement the change requests, and some of them may be executed at the same time.

Let CR be the set of the change requests implemented by the CSWs in a CSE: $CR = \{..., cr_i, ...\}$.

Each change request cr_i is associated with a csw_i in the CSE: $csw_i = GCSW(cr_i) =< id_i, A_i, E_i, D_i, C_i, W_i, GD_i, GC_i, GW_i, GT_i >$, in which A_i is the set of change activities of csw_i: $A_i = \{..., a_{ik}, ...\}$.

An activity a_{ik} in A_i can apply many changes to many artifacts: $GC_i(a_{ik}) = \{c_{ik1}, c_{ik2}, ...\}$.

A change of an activity will *write* (*add*, *delete*, or *modify*) an artifact with refer to some other artifacts, *read* artifacts.

For example, $GD_i(c_{ik1}) = \{d_{ik11}, d_{ik12}, d_{ik13}\}$ means that the change c_{ik1} of the activity a_{ik} uses, *reads*, the artifacts d_{ik12}, d_{ik13} to *write* the artifact d_{ik11}. In other words, $GD_i(c_{ik1}) = GDT_i(c_{ik1}, w) \cup GDT_i(c_{ik1}, r)$. For the ease of understand, we write $GD_i(c_{ik1}) = \{w(d_{ik11}), r(d_{ik12}, d_{ik13})\}$ (r: ***r****ead*, w: ***w****rite*).

Because artifacts in a software system are not separated but may be connected by dependency relationships, changes to some artifacts in a CSW can unexpectedly affect to changes in other CSWs.

Let us consider two CSWs csw_i and csw_j in the CSE.

- $\mathcal{GCSW}(cr_i) = csw_i =< id_i, A_i, E_i, D_i, C_i, W_i, GD_i, GC_i, GW_i, GT_i >$ with
  $A_i = \{..., a_{ik}, ...\}$, $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$, $GD_i(c_{ikm}) = \{w(d_{ikm1}), r(d_{ikm2}, ...)\}$.
- $\mathcal{GCSW}(cr_j) = csw_j =< id_j, A_j, E_j, D_j, C_j, W_j, GD_j, GC_j, GW_j, GT_j >$ with
  $A_j = \{..., a_{jl}, ...\}$, $GC_j(a_{jl}) = \{..., c_{jln}, ...\}$, $GD_j(c_{jln}) = \{w(d_{jln1}), r(d_{jln2}, ...)\}$.

In the case of $d_{ik...} = d_{jl...}$ or $d_{ik...}$ depending on $d_{jl...}$, an inconsistency may happen if the worker $w_{ik} = GW(a_{ik})$, who applies changes to $d_{ik...}$, does not recognize the changes or the impact of the changes on $d_{jl...}$ of another worker $w_{jl} = GW(a_{jl})$, or the worker w_{jl} does not recognize the changes on $d_{ik...}$ of the worker w_{ik}, or the impact of his changes on the changes on $d_{ik...}$ of the worker w_{ik}.

Assuming that a worker will synchronize his workspace with the remote repository when he starts an activity, and will commit his changes when he finishes the activity. $CO(a_{ik}, d)$ denotes the version of the artifact d when it is checked-out at the beginning of the activity a_{ik}. $CI(a_{ik}, d)$ denotes the updated version of the artifact d when it is checked-in at the end of the activity a_{ik}.

Previous work monitored the ongoing changes and reported a (potential) conflict, a type of inconsistencies, if $c_{ik..}$ and $c_{jl...}$ are the concurrent changes and $d_{ik...} = d_{jl...}$ or $d_{ik...}$ depends on $d_{jl...}$, with $GD_i(c_{ik..}) = \{w(d_{ik...}), r(...)\}$ and $GD_j(c_{jl..}) = \{w(d_{jl...}), r(...)\}$.

Differently from the previous work, we pay attention to both concurrent and non-concurrent changes and the context of a change. A CSW provides the details of the change history, current changes, and likely changes of a change inside it. Therefore, the CSW containing a change is considered as the context of the change. For example, the context of the change $c_{ik...}$ is the CSW csw_i containing the activity a_{ik} with $GC_i(a_{ik}) = \{c_{ik...}, ...\}$. The context of the change $c_{jl...}$ is the CSW csw_j containing the activity a_{jl} with $GC_j(a_{jl}) = \{c_{jl...}, ...\}$.

- $Context(a_{ik}) = csw_i$, $Context(c_{ik...}) = \{a_{ik}, w_{ik}, csw_i\}$
- $Context(a_{jl}) = csw_j$, $Context(c_{jl...}) = \{a_{jl}, w_{jl}, csw_j\}$

We can add or remove some information from the change context according to the user's customization.

For example, $Context(a_{ik}) = \{cr, cr_i, csw_i\}$, $Context(c_{ik...}) = \{a_{ik}, w_{ik}, csw_i, cr_i\}$.

When the previous work reported a conflict, they just provided the workers with the information of the involved changes only. On the other hand, we report an inconsistency with the information about the contexts of the related changes. For instance, to report a conflict $Conflict_1$ relating to the changes $c_{ik...}$ and $c_{jl...}$:

- The previous works provided $Context(Conflict_1) = \{c_{ik...}, w_{ik}, c_{jl...}, w_{jl}\}$.
- We provide $Context(Conflict_1) = \{c_{ik...}, a_{ik}, w_{ik}, csw_i, c_{jl...}, a_{jl}, w_{jl}, csw_j\}$.

4.3.1 Intra-Direct-Conflict

An Intra-Direct-Conflict happens if there exist two activities, a_{ik} and a_{it}, in the same CSW, csw_i, where:

- a_{ik} happens concurrently with a_{it}: $[S(a_{ik}), F(a_{ik})] \cap [S(a_{it}), F(a_{it})] \neq \emptyset$.

- $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$ with $GD_i(c_{ikm}) = \{w(d), r(...)\}$.
- $GC_i(a_{it}) = \{..., c_{itn}, ...\}$ with $GD_i(c_{itn}) = \{w(d), r(...)\}$.
- a_{ik} and a_{it} change the same version of d: $CO(a_{ik}, d) = CO(a_{it}, d)$.

Context(Intra-Direct-Conflict) $= \{c_{ikm}, a_{ik}, w_{ik}, c_{itn}, a_{it}, w_{it}, csw_i\}$.

4.3.2 Inter-Direct-Conflict

An Inter-Direct-Conflict happens if there exist two activities, a_{ik} and a_{jl}, in different CSWs, csw_i and csw_j, where:

- a_{ik} happens concurrently with a_{jl}: $[S(a_{ik}), F(a_{ik})] \cap [S(a_{jl}), F(a_{jl})] \neq \emptyset$.
- $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$ with $GD_i(c_{ikm}) = \{w(d), r(...)\}$.
- $GC_j(a_{jl}) = \{..., c_{jln}, ...\}$ with $GD_j(c_{jln}) = \{w(d), r(...)\}$.
- a_{ik} and a_{jl} change the same version of d: $CO(a_{ik}, d) = CO(a_{jl}, d)$.

Context(Inter-Direct-Conflict) $= \{c_{ikm}, a_{ik}, w_{ik}, csw_i, c_{jln}, a_{jl}, w_{jl}, csw_j\}$.

4.3.3 Intra-Indirect-Conflict

An Intra-Indirect-Conflict happens if there exist two activities, a_{ik} and a_{it}, in the same CSW, csw_i, where:

- a_{ik} happens concurrently with a_{it}: $[S(a_{ik}), F(a_{ik})] \cap [S(a_{it}), F(a_{it})] \neq \emptyset$.
- $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$ with $GD_i(c_{ikm}) = \{w(d), r(...)\}$.
- $GC_i(a_{it}) = \{..., c_{itn}, ...\}$ with $GD_i(c_{itn}) = \{w(d'), r(d, ...)\}$.
- a_{ik} changes a version of d that is read by a_{it} to change d': $CO(a_{ik}, d) = CO(a_{it}, d)$.
- The change c_{ikm} on d of a_{ik} affects the change c_{itn} on d' of a_{it}.

Context(Intra-Indirect-Conflict) $= \{c_{ikm}, a_{ik}, w_{ik}, c_{itn}, a_{it}, w_{it}, csw_i\}$.

4.3.4 Inter-Indirect-Conflict

An Inter-Indirect-Conflict happens if there exist two activities, a_{ik} and a_{jl}, in different CSWs, csw_i and csw_j, where:

- a_{ik} happens concurrently with a_{jl}: $[S(a_{ik}), F(a_{ik})] \cap [S(a_{jl}), F(a_{jl})] \neq \emptyset$.
- $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$ with $GD_i(c_{ikm}) = \{w(d), r(...)\}$.
- $GC_j(a_{jl}) = \{..., c_{jln}, ...\}$ with $GD_j(c_{jln}) = \{w(d'), r(d, ...)\}$.
- a_{ik} changes a version of d that is read by a_{jl} to change d': $CO(a_{ik}, d) = CO(a_{jl}, d)$.
- The change c_{ikm} on d of a_{ik} affects the change c_{jln} on d' of a_{jl}.

Context(Inter-Indirect-Conflict) $= \{c_{ikm}, a_{ik}, w_{ik}, csw_i, c_{jln}, a_{jl}, w_{jl}, csw_j\}$.

4.3.5 Direct-Revision-Inconsistency

A Direct-Revision-Inconsistency happens if there exist two activities, a_{ik} and a_{jl}, in different CSWs, csw_i and csw_j, where:

- a_{ik} precedes a_{jl}: $F(a_{ik}) < S(a_{jl})$.
- $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$ with $GD_i(c_{ikm}) = \{w(d), r(...)\}$.
- $GC_j(a_{jl}) = \{..., c_{jln}, ...\}$ with $GD_j(c_{jln}) = \{w(d), r(...)\}$.
- a_{jl} changes the version of d created before by a_{ik}: $CI(a_{ik}, d) = CO(a_{jl}, d)$.
- The change c_{jln} on d of a_{jl} contradicts the change c_{ikm} on d of a_{ik}. The worker w_{jl} misunderstands the purpose of the worker w_{ik} in making the change c_{ikm}.

Context(Direct-Revision-Inconsistency) $= \{c_{ikm}, a_{ik}, w_{ik}, csw_i, c_{jln}, a_{jl}, w_{jl}, csw_j\}$.

4.3.6 Indirect-Revision-Inconsistency

An Indirect-Revision-Inconsistency happens if there exist two activities, a_{ik} and a_{jl}, in different CSWs, csw_i and csw_j, where:

- a_{ik} precedes a_{jl}: $F(a_{ik}) < S(a_{jl})$.
- $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$ with $GD_i(c_{ikm}) = \{w(d'), r(d, ...)\}$.
- $GC_j(a_{jl}) = \{..., c_{jln}, ...\}$ with $GD_j(c_{jln}) = \{w(d), r(...)\}$.
- a_{jl} changes a version of d that is read by a_{ik} to change d': $CO(a_{ik}, d) = CO(a_{jl}, d)$.
- The change c_{jln} on d of a_{jl} affects the change c_{ikm} on d' of a_{ik}.
- The worker w_{jl} does not recognize the impact of his change, c_{jln}, on the previous change, c_{ikm}, of w_{ik}: $\nexists\ c_{jl...} \in GC_j(a_{jl})$ so that $GD_j(c_{jl...}) = \{w(d'), r(d, ...)\}$.

Context(Indirect-Revision-Inconsistency) $= \{c_{ikm}, a_{ik}, w_{ik}, csw_i, c_{jln}, a_{jl}, w_{jl}, csw_j\}$.

4.3.7 RWR Interleaving-Inconsistency

A RWR Interleaving-Inconsistency happens if there exist three activities, a_{ik}, a_{jl}, and a_{it}, where:

- a_{ik} and a_{it} are in the same CSW, csw_i, and a_{jl} is in a different CSW, csw_j.
- a_{jl} happens before a_{it} but after a_{ik}: $[S(a_{jl}), F(a_{jl})] \subset [F(a_{ik}), S(a_{it})]$.
- $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$ with $GD_i(c_{ikm}) = \{w(d'), r(d, ...)\}$.
- $GC_j(a_{jl}) = \{..., c_{jln}, ...\}$ with $GD_j(c_{jln}) = \{w(d), r(...)\}$.
- $GC_i(a_{it}) = \{..., c_{ito}, ...\}$ with $GD_i(c_{ito}) = \{w(d"), r(d, ...)\}$.

- a_{jl} changes a version of d that is read by a_{ik} to change d', $CO(a_{ik}, d) = CO(a_{jl}, d)$.
- The version of d used by a_{it} to change d" is the version created by a_{jl}, $CI(a_{jl}, d) = CO(a_{it}, d)$. However, the worker w_{it} does not recognize the change c_{jln} of a_{jl}. He treats the new version in the same way as the old version used by a_{ik}.
- The change c_{jln} on d of a_{jl} affects the change c_{ikm} on d' of a_{ik} and the change c_{ito} on d" of a_{it} through the change on semantics or syntax of d.

Context(RWR Interleaving-Inconsistency) = $\{c_{ikm}, a_{ik}, w_{ik}, c_{ito}, a_{it}, w_{it}, csw_i, c_{jln}, a_{jl}, w_{jl}, csw_j\}$.

4.3.8 WWR Interleaving-Inconsistency

A WWR Interleaving-Inconsistency happens if there exist three activities, a_{ik}, a_{jl}, and a_{it}, where:

- a_{ik} and a_{it} are in the same CSW, csw_i, and a_{jl} is in a different CSW, csw_j.
- a_{jl} happens before a_{it} but after a_{ik}: $[S(a_{jl}), F(a_{jl})] \subset [F(a_{ik}), S(a_{it})]$.
- $GC_i(a_{ik}) = \{..., c_{ikm}, ...\}$ with $GD_i(c_{ikm}) = \{w(d), r(d, ...)\}$.
- $GC_j(a_{jl}) = \{..., c_{jln}, ...\}$ with $GD_j(c_{jln}) = \{w(d), r(...)\}$.
- $GC_i(a_{it}) = \{..., c_{ito}, ...\}$ with $GD_(c_{ito}) = \{w(d'), r(d, ...)\}$.
- a_{jl} changes the version of d that is created by a_{ik}, $CI(a_{ik}, d) = CO(a_{jl}, d)$.
- The version of d used by a_{it} to change d' is the version created by a_{jl}, $CI(a_{jl}, d) = CO(a_{it}, d)$. However, the worker w_{it} does not recognize the change c_{jln} of a_{jl}. He treats the new version in the same way as the old version created by a_{ik}.
- The change c_{jln} on d of a_{jl} contradicts the change c_{ikm} on d of a_{ik}, and affects the change c_{ito} on d' of a_{it} through the change on semantics or syntax of d.

Context(WWR Interleaving-Inconsistency) = $\{c_{ikm}, a_{ik}, w_{ik}, c_{ito}, a_{it}, w_{it}, csw_i, c_{jln}, a_{jl}, w_{jl}, csw_j\}$.

4.4 Summary

Presenting in a formal way the main concepts used in our approach, such as CSW, CSE, and the patterns of inconsistency, lays the foundation for our research, and enhances further developments based on this theoretical model.

Chapter 5

Inconsistency Awareness

This chapter describes in detail our approach to detecting the inconsistencies in collaborative software development in real time, and how to implement it. We also suggest some solutions to the detected inconsistencies. In addition, we present a formal method for modeling the main behaviors of CSE in CP-nets and verifying the generated model to detect inconsistencies.

5.1 Possibility of Inconsistency

To detect an inconsistency, we need to identify the necessary condition and sufficient condition of an inconsistency based on the patterns of inconsistency. The explicit properties relating to the relationships between the artifacts, the time orders of the activities applying the changes to the artifacts, and the contexts of the changes, can be considered as the necessary conditions for the occurrence of an inconsistency. In other words, these properties may, not must, lead to an inconsistency. Except for the inconsistencies related to syntactic errors, for example the change of a method's signature that another method calls in the case of an indirect-conflict, the sufficient condition of an inconsistency depends on the consistency between the semantics of the changes and the intention of the workers who made the changes. Because this consistency can only be decided by the workers, their decisions are the final judge.

Therefore, we can detect exactly the situations that may lead to an inconsistency, namely *potential inconsistency*, by using the mentioned explicit properties of the patterns of inconsistency. Regarding inconsistency relating to syntactic errors, we can determine whether the detected potential inconsistency will result in a real inconsistency or not by analyzing more deeply the syntactic properties of the detected potential inconsistency. In the case of semantic errors, the workers, who implement the changes with their own intention, will decide if the reported potential inconsistency will actually lead to an inconsistency.

To support inconsistency detection, we specify the potential cases of the patterns of inconsistency based on the necessary conditions of the pattern of inconsistency (the relationships of the artifacts, the time orders of the activities applying the changes to the artifacts, and the contexts of the changes). In the case of Intra-Direct-Conflict and Inter-Direct-Conflict, these necessary conditions are also the sufficient conditions. Therefore, there are no potential cases of these patterns.

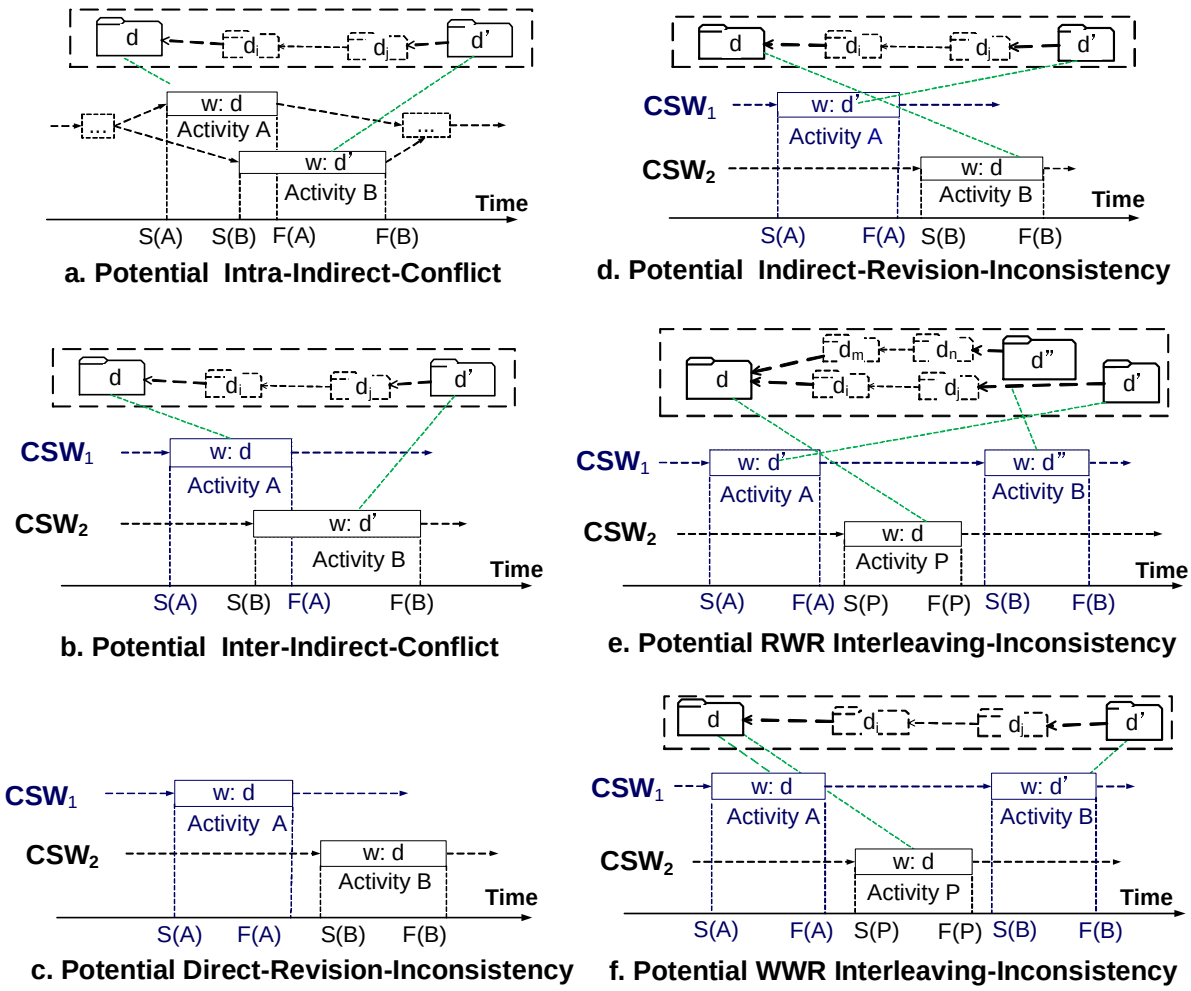


Figure 5.1: Potential inconsistency

5.1.1 Potential Intra-Indirect-Conflict

A situation in which concurrent activities in the same CSW change (*write*) different artifacts that are connected by dependency relationships.

Fig. 5.1a describes a potential Intra-Indirect-Conflict in which two concurrent activities A and B in the same CSW change the artifacts d and d', respectively:

- A and B are concurrent: $[S(A), F(A)] \cap [S(B), F(B)] \neq \emptyset$;
- d' can reach d by dependency relationships.

5.1.2 Potential Inter-Indirect-Conflict

A situation in which concurrent activities in different CSWs change (*write*) different artifacts that are connected by dependency relationships.

Fig. 5.1b describes a potential Inter-Indirect-Conflict in which two concurrent activities A and B in different CSWs change the artifacts d and d', respectively:

- A and B are concurrent: $[S(A), F(A)] \cap [S(B), F(B)] \neq \emptyset$;
- d' can reach d by dependency relationships.

5.1.3 Potential Direct-Revision-Inconsistency

A situation in which a later change to an artifact contradicts with the previous changes to the same artifact, made by tasks in different CSWs.

Fig. 5.1c describes a potential Direct-Revision-Inconsistency in which two non-concurrent activities A and B in different CSWs change two successive versions of the artifact d respectively:

- A happens before B: $F(A) < S(B)$;
- B ***w**rites* the version created by A: $CO(B, d) = CI(A, d)$ and $CI(B, d) \neq CI(A, d)$.

5.1.4 Potential Indirect-Revision-Inconsistency

A situation in which an activity changes (***w**rites*) an artifact d which some artifacts, that were changed (***w**riten*) before by the earlier tasks in different CSWs, can reach by dependency relationships.

Fig. 5.1d describes a potential Indirect-Revision-Inconsistency in which two non-concurrent activities A and B in different CSWs change the artifacts d and d', respectively:

- A happens before B: $F(A) < S(B)$;
- d' can reach d by dependency relationships.

5.1.5 Potential RWR Interleaving-Inconsistency

A situation in which two activities A and B in the same CSW sequentially change (***w**rite*) the artifacts d' and d", respectively, and an activity P in another change process changes (***w**rites*) an artifact d, which d' and d" can reach by dependency relationships, at sometime during the interval between A and B.

Fig. 5.1e describes a potential RWR Interleaving-Inconsistency in which three activities A, P, and B change the artifacts d', d, and d", respectively, and two tasks A and B are in the same CSW that is different from the CSW of P.

- P happens after A and before B: $[S(P), F(P)] \subset [F(A), S(B)]$;
- d' and d" can reach d by dependency relationships.

5.1.6 Potential WWR Interleaving-Inconsistency

A situation in which two activities A and B in the same CSW sequentially change (***write***) the artifacts d and d', respectively, and an activity P in a different CSW changes (***writes***) d at sometime during the interval between A and B. d' can reach d by dependency relationships.

Fig. 5.1f describes a potential WWR Interleaving-Inconsistency in which three activities A, P, and B change the artifacts d, d, and d', respectively, and two activities A and B are in the same CSW that is different from the CSW of P:

- P happens after A and before B: $[S(P), F(P)] \subset [F(A), S(B)]$;
- P ***writes*** the version created by A: $CO(P, d) = CI(A, d)$ and $CI(P, d) \neq CI(A, d)$.
- d' can reach d by dependency relationships.

5.2 Inconsistency Detection in Real Time

5.2.1 Approach

Our approach is based on two hypotheses:

- Although detecting emerging inconsistency can produce false warnings, the cost of examining these warnings is still much lower than the cost of fixing the defects that are caused by the inconsistencies detected late in the software life cycle.
- Information about the preceding changes, succeeding changes, and concurrent changes of the changes causing a (potential) inconsistency reminds the workers involved of the contexts in which their changes were made. This helps them understand the inconsistency easily and correctly, and resolve it more effectively.

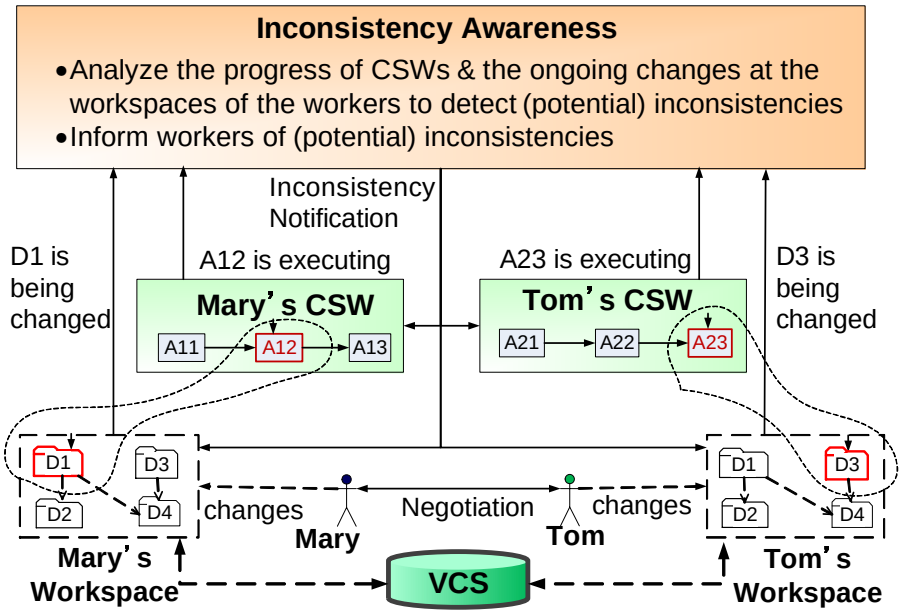


Figure 5.2: Inconsistency awareness approach overview

To obtain the latest information about the ongoing changes in the system, we use the workspace awareness technique to collect information about the ongoing changes in the workspaces of the workers, share this information across their workspaces, and alert them of emerging conflicts. To reduce false warnings, we apply the fine-grained analysis to identify structured changes at the client workspaces at the level of program entities. However, because the changes between a check-out and a check-in may be applied to the same program entity or negate each other, we transform the textual changes in each commit to a VCS into the structured changes, and consider the output as the compact set of the changes made in the workspace of a worker between a check-out and a check-in associated with the commit.

In order to detect other types of inconsistency besides conflicts and resolve inconsistency more easily, we combine the workspace awareness technique with the context awareness technique. Context awareness in the scope of this dissertation means sharing information about the context of a change. As a change process can reveal much information about a change inside it, for example, the preceding changes, the concurrent changes, the succeeding changes, and the change purpose, we consider the CSW representing the change process as the context of that change.

Fig. 5.2 shows an overview of our inconsistency awareness approach. Monitoring the progress of CSWs (context awareness), such as the start time and finish time of an activity in a CSW, helps obtain the context of a change in the system. Monitoring the changes at client workspaces (workspace awareness), such as renaming an existing method or adding a new method, helps detect new changes immediately. Analyzing the latest information about the changes in the system and their contexts allows us to 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 showed the workers the concurrent changes causing a (potential) conflict only.

5.2.2 Information Preparation for Inconsistency Detection

Fig. 5.3 shows the structure of a CSW with the necessary information to detect (potential) inconsistencies. There may be many *atomic changes* happening in the workspace of a worker during the execution time $[S(A), F(A)]$ of an activity A in his CSW, CSW_n. $S(A)$ and $F(A)$ denote the start time, S, and the 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 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*, the being changed artifact, *inbound artifacts*, the artifacts depending on the main artifact, and the *outbound artifacts*, the artifacts on which the main artifact depends.

The inbound and outbound artifacts of a main artifact are found by examining the dependency relationships starting from or ending at the main artifact. Dependency is a relationship that states that an artifact A uses the information and service of another artifact B, but not necessarily the reverse [4]. We say that A depends on B.

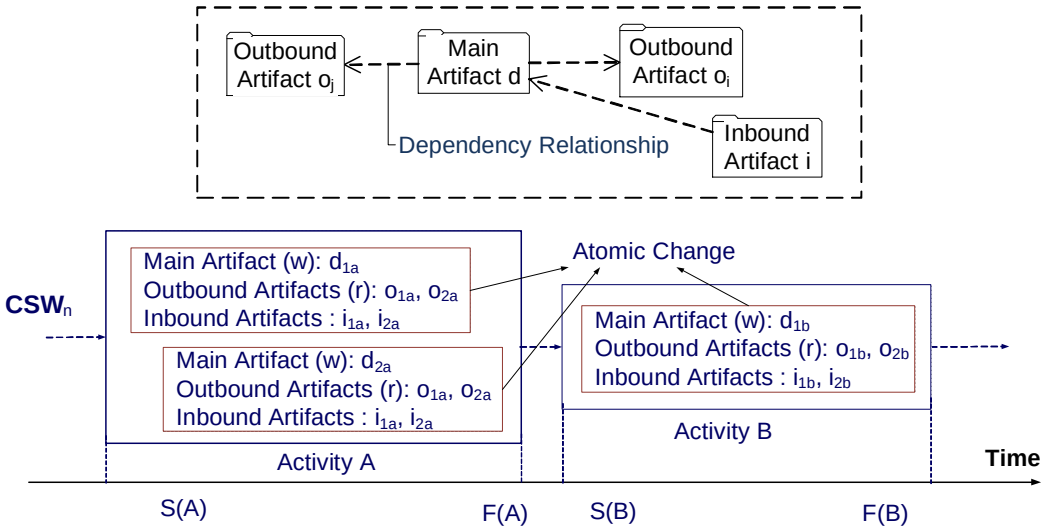


Figure 5.3: A simplified structure of a Change Support Workflow

Artifacts are packages, classes, and features that can be class attributes, constructors, and methods. There are two main types of the relationship [35]:

- The first type of relationship is *composition*. Packages have classes, which themselves have features.
- The second type of relationship is *dependency*. Classes refer to each other, features refer to each other, and features refer to classes.

A dependency can be either *direct* or *indirect* dependency. If A depends on B, then A has a direct dependency with B. If A depends on B, B depends on C, then A has an indirect dependency with C. In other words, A has an N-level dependency with B where N is the number of intermediate dependencies between A and B. N-level dependency is a direct dependency if $N = 1$, and is an indirect dependency if $N > 1$.

5.2.3 Inconsistency Detection Procedure

As we have mentioned in Section 5.1, it is very difficult to represent the semantic aspect of an inconsistency in a formal way. Here we present the procedure to detect the situations that may lead to the patterns of inconsistency, presented in Chapter 3.2, based on the Intra-Direct-Conflict and Inter-Direct-Conflict patterns and the potential cases of the remaining patterns (Section 5.1). Except for direct conflicts and the inconsistencies related to syntactic errors, the workers need to confirm the correctness of the detected situation.

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

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), *csw* (the CSW containing the activity), *project* (the project containing the artifacts), *author* (the worker making the change), and *version* (the version of the main artifact assigned by a VCS).

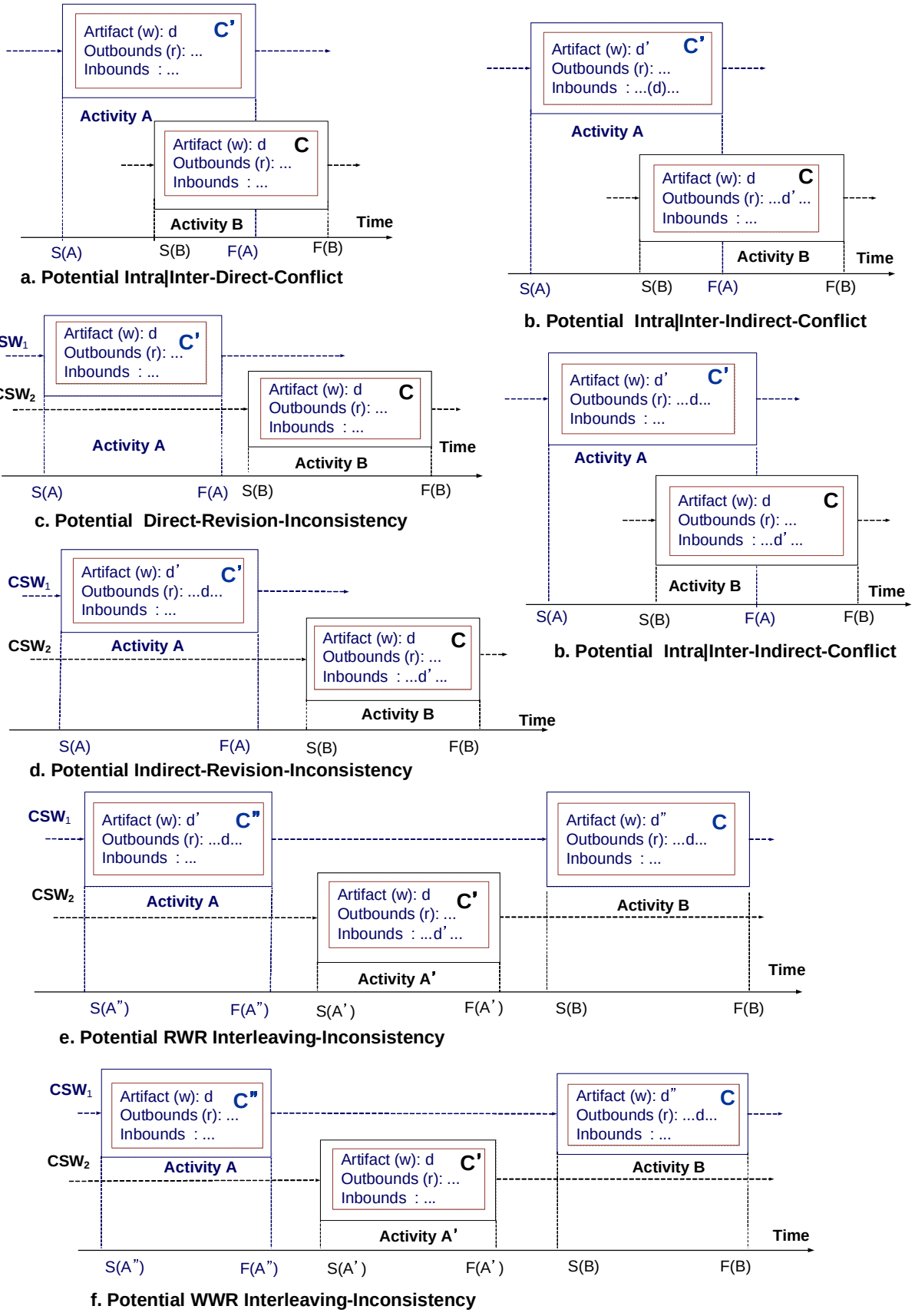
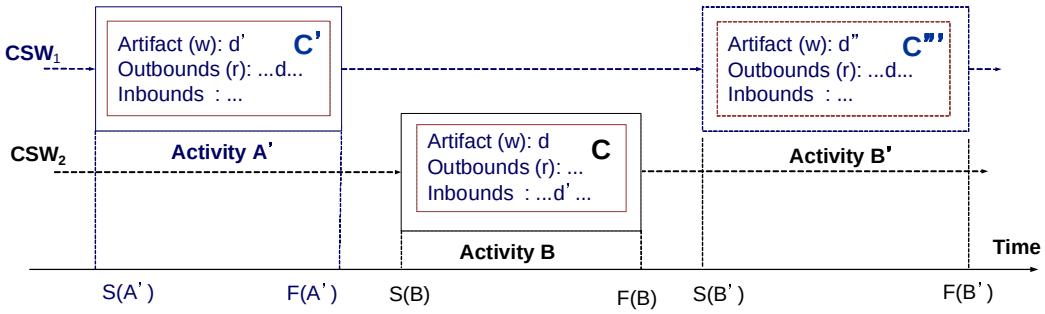
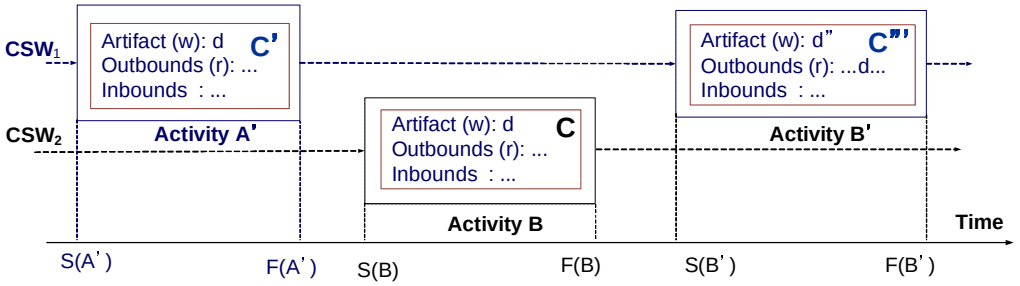


Figure 5.4: Illustration of inconsistency detection



e'. Potential RWR Interleaving-Inconsistency in plan



f'. Potential WWR Interleaving-Inconsistency in plan

Figure 5.5: Illustration of inconsistency detection (con't)

2. Search the database for the atomic changes C' applied to the artifacts in the same project with C but made by other workers, i.e. $C'.project = C.project$, $C'.author \neq C.author$, and the progress of their CSWs.

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. 5.4a):
     $(C'.artifact = C.artifact)$ AND
     $(C'.version = C.version)$ AND
     $([C.activity.startTime, C.activity.finishTime] \cap$
     $[C'.activity.startTime, C'.activity.finishTime] \neq \emptyset)$
     - If C' and C are in the same CSW ($C'.csw = C.csw$), report an **Intra-Direct-Conflict**.
     - If C' and C are in different CSWs ($C'.csw \neq C.csw$), report an **Inter-Direct-Conflict**.
     - If C' is a change that has not been made yet, i.e. a planned change, report to the worker of C' a **potential Intra-Direct-Conflict in plan**, if C' and C are in the same CSW, or a **potential Inter-Direct-Conflict in plan**, if C' and C are in different CSWs.
   - 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. 5.4b):

($(C.inbounds \supset C'.artifact)$ OR $(C.outbounds \supset C'.artifact)$) AND
$([C.activity.startTime, C.activity.finishTime] \cap$
$[C'.activity.startTime, C'.activity.finishTime] \neq \emptyset)$
- If C' and C are in the same CSW ($C'.csw = C.csw$), report a **potential Intra-Indirect-Conflict**.
- If C' and C are in different CSWs ($C'.csw \neq C.csw$), report a **potential Inter-Indirect-Conflict**.
- If C' is a change that has not been made yet, i.e. a planned change, report to the worker of C' an **Intra-Indirect-Conflict in plan**, if C' and C are in the same CSW, or an **Inter-Indirect-Conflict in plan**, if C' and C are in different CSWs.

- Report a **potential Direct-Revision-Inconsistency** between C and C', if C' is in a different CSW and 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. 5.4c):
  $(C'.csw \neq C.csw)$ AND
  $(C'.artifact = C.artifact)$ AND
  $(C'.version \leq C.version)$ AND
  $(C'.activity.finishTime < C.activity.startTime)$
- Report a **potential Indirect-Revision-Inconsistency** between C and C', if C' is in a different CSW and has the main artifact d' depending on the main artifact d of C, and $C'.activity$ happens before $C.activity$ (Fig. 5.4d):
  $(C'.csw \neq C.csw)$ AND
  $(C'.outbounds \supset C.artifact)$ AND
  $(C'.activity.finishTime < C.activity.startTime)$
- **Potential RWR Interleaving-Inconsistency** (Fig. 5.4e)
  For each artifact d in $C.outbounds$:
  - Find an atomic change C' that is in a CSW different from that of C and has d as the main artifact. $C'.activity$ happens before $C.activity$:
    $(C'.csw \neq C.csw)$ AND
    $(C'.artifact = d)$ AND
    $(C'.activity.finishTime < C.activity.startTime)$
  - Find an atomic change C" that is in the same CSW with C and has the main artifact depending on d. C".$activity$ happens before $C.activity$:
    $(C".csw = C.csw)$ AND
    $(C".outbounds \supset d)$ AND
    $(C".activity.finishTime < C.activity.startTime)$
  - If C".$activity$ happens before $C'.activity$ ($C".activity.finishTime < C'.activity.startTime$), report a **potential RWR Interleaving-Inconsistency** among $C", C', C$ on d.
- **Potential WWR Interleaving-Inconsistency** (Fig. 5.4f)
  For each artifact d in $C.outbounds$:

- Find an atomic change C' that is in a CSW different from that of C and has d as the main artifact. $C'.activity$ happens before $C.activity$:

($C'.csw \neq C.csw$) AND

($C'.artifact = d$) AND

($C'.activity.finishTime < C.activity.startTime$)

- Find an atomic change C" that is in the same CSW with C and has d as the main artifact. C".$activity$ happens before $C.activity$:

(C".$csw = C.csw$) AND

(C".$artifact = d$) AND

(C".$activity.finishTime < C.activity.startTime$)

- If C".$activity$ happens before $C'.activity$ (C".$activity.finishTime < C'.activity.startTime$), report a **potential WWR Interleaving-Inconsistency** among C", C', C on d.

- **Potential RWR Interleaving-Inconsistency in plan** (Fig. 5.5e')

  - Find an atomic change C' that is in a CSW different from that of C and has the main artifact d' depending on the main artifact d of C. $C'.activity$ happens before $C.activity$:

  ($C'.csw \neq C.csw$) AND

  ($C'.outbounds \supset C.artifact$) AND

  ($C'.activity.finishTime < C.activity.startTime$)

  - Find an atomic change $C"'$, which is a planned change in the same CSW with C' and depends on d. $C"'.activity$ has not been executed yet:

  ($C"'.csw = C'.csw$) AND

  (($C"'.outbounds \supset d$) OR ($C.inbounds \supset C"'.artifact$)) AND

  ($C"'.activity.startTime = 0$)

  - Report a **potential RWR Interleaving-Inconsistency in plan** among $C', C, C"'$ on d.

- **Potential WWR Interleaving-Inconsistency in plan** (Fig. 5.5f')

  - Find an atomic change C' that is in a CSW different from that of C and has d as the main artifact. d is the main artifact of C too. $C'.activity$ happens before $C.activity$:

  ($C'.csw \neq C.csw$) AND

  ($C'.artifact = d$) AND

  ($C'.activity.finishTime < C.activity.startTime$)

  - Find an atomic change $C"'$, which is a planned change in the same CSW with C' and depends on d. $C"'.activity$ has not been executed yet:

  ($C"'.csw = C'.csw$) AND

  (($C"'.outbounds \supset d$) OR ($C.inbounds \supset C"'.artifact$)) AND

  ($C"'.activity.startTime = 0$)

  - Report a **potential WWR Interleaving-Inconsistency in plan** among $C', C, C"'$ on d.

5.2.4 Inconsistency Resolution

Upon receiving a warning of a (potential) inconsistency, one should examine the context of the reported inconsistency to solve it or skip it in the case of a false warning.

Resolving (potential) inconsistencies relating to semantic errors is challenging, because different workers design different CSWs for different change requests with their own intentions. If a worker is not sure about the purpose of other workers after examining their CSWs and the contents of their changes, he should contact them to conduct a negotiation. In this case, the cooperation of the workers is the most important factor. Face-to-face discussion, email, phone, instant messenger, etc. can be their communication means.

Regarding inconsistency relating to planned CSWs, the following methods can be considered:

- If potential inconsistencies are reported between a planned CSW and executing CSWs, the planned CSW can reorder its activities so that its inconsistency-related activities can be delayed until the inconsistency-related activities of the executing CSWs are finished.

- If potential inconsistencies are reported between the planned CSWs or the planned part of CSWs, their change requests should be combined and redivided to reduce the coupling among the artifacts affected by these CSWs.

As for inconsistency relating to executing CSWs, the worker who makes the later changes leading to a potential inconsistency should reconsider the relevance of his changes first. In case he still keeps his opinion, he should contact the workers involved to persuade them to update their affected artifacts to adapt to his changes.

- **Intra|Inter-Direct-Conflict**: The later worker can delay his current changes until the first worker finishes her changes to the artifact. Then he can obtain the updated version and change it. If he thinks that his intention contradicts hers, he should contact the first worker, rather than waiting or switching to other activity.

- **Intra|Inter-Indirect-Conflict**: The first worker, whose artifact depends on the artifact changed by the second worker, should delay her current changes or not commit her changes, until the second worker finishes his changes. Then, she can obtain the updated version, examine the differences, and use it. The second worker should support the first worker by notifying her of his intention when making these changes directly, or through the comment part of his corresponding activity.

- **Direct-Revision-Inconsistency**: The second worker should examine the context of the inconsistency carefully to make sure that he understands the intention of the first worker correctly. He can contact her in case the information supplied by our system is not clear enough. The second worker should also execute the test suit of the first worker, if he still conducts his change.

- **Indirect-Revision-Inconsistency**: If the second worker keeps his opinion, he can conduct his changes and then, updates the changes of the first worker impacted by his changes to ensure the consistency of the effects by his changes. He can also send some extra messages to the first worker, or describes more detail his change purpose in the comment part of the corresponding activity.

- **RWR Interleaving-Inconsistency**: This warning only appears if the warning about Indirect-Revision-Inconsistency before is not viewed by both workers. The first worker should examine the changes of the second worker, and adjusts her current changes and her past changes to fit his changes.
- **WWR Interleaving-Inconsistency**: This warning only appears if the warning about Direct-Revision-Inconsistency before is not viewed by both workers. The first worker should adapt her current changes to the changes of the second worker. In case she does not agree with the changes of the second worker, a negotiation should be conducted.

5.3 A Formal Method to Detect Inconsistency

This section presents a method for modeling the essential behaviors of CSE and analyzing the formal model to detect data abnormalities, specially the patterns of inconsistency. First, we give an overview of the method. Next we introduce the related work in workflow modeling and verification, followed by an explanation of the basic concepts of Colored Petri Nets that are used to model CSE. Finally, we describe our modeling and verification method along with the illustrating examples.

5.3.1 Overview

As described in the previous chapter, inconsistencies are directly related to data, therefore, data-flow verification is required in CSE. However, most previous works in workflow verification concentrated on structure verification, temporal verification, and resource verification. Although data is an important aspect of workflows, only little research in data-flow verification can be observed and they focused on data-flow errors in a single workflow instance. Unlike previous work, we make the following contributions:

- Using Colored Petri Nets (CP-nets or CPN) to model workflows instead of extending existing languages to represent the data factor like the works in [49, 50, 52, 53].
- Being able to represent data and changes on the properties of data explicitly.
- Being able to represent both control flow and data flow in one single model.
- Considering the data-flow errors caused by mutual influences among the data-related workflows, in addition to the interactions among concurrent activities in a single workflow.
- Our method can be applied to model and verify data-related abnormalities of other types of workflows.

We use CP-nets to model the necessary behaviors of CSE because:

1. CP-nets combine the capabilities of Petri Net, a basic model widely used for modeling and analyzing control-flow of workflows, with the high-level language.
2. We can use CPN Tools [42] to edit, simulate, and analyze the CPN model of CSE.

3. We can avoid defining a new language by extending the existing languages to represent unsupported factors like [49, 50, 52, 53], and avoid the need of proving the correctness of the new language like [55] as well.

4. Aspects of a CSW including control (activity, edge), data (artifact), resource (worker), and time (Execution time) can be represented by CP-nets. However, because modeling resources were mentioned in some previous studies [44, 45, 46, 47], we ignore modeling resources to reduce the complexity of the generated model.

    - Allowing tokens to be associated with colors in CP-nets helps us represent the changes on the properties of data, for example, version and content, during the execution of CSWs easily.
    - CP-nets include the *time concept* which helps us specify the execution time of activities in a CSW.
    - CP-nets support the *hierarchy concept* which allows us to model large systems easily.

5.3.2 Related Work in Data-Flow Modeling and Verification

Workflow verification has received a lot of attention, specially structure verification, resource verification, and temporal verification [44, 45, 46, 47]. However, most verification techniques ignore data aspect and there is little support for data-flow verification. In addition, although there are many ways to model a workflow, such as directed graph, Business Process Model and Notation (BPMN), WF-Net [43], UML activity diagram, they do not specify data flow formally or their specifications do not help with data-flow verification. Therefore, suitable data modeling must be conducted before data verification.

Reference [48] was one of the first studies, which mentioned the importance of data-flow verification, and identified possible errors in data flow, for instance, missing data, redundant data, and conflicting data. Some general discussions on data-flow modeling, specifications and verifications were given, but without any detailed solution. In [49], the authors used a data-flow matrix and an extension of the UML activity diagram that incorporates data input and output, to specify data. Then, a dependency-based algorithm was proposed to detect three basic types of data-flow anomalies: missing data, redundant data, and conflicting data. Reference [50] proposed another workflow modeling technique, named Dual Workflow Nets, which enabled describing both data flow and control flow by introducing new types of nodes and distinguishing data token from control token. A formal verification method for detecting the control/data-flow inconsistencies was also presented. A graph traversal approach was used in [51] to build an algorithm for detecting lost data, missing data, and redundant data. Another approach for modeling the data flow of BPMN was given in [52]. The authors formalized the basic data object processing in BPMN using Petri nets, and presented a technique for repairing some data anomalies corresponding to deadlocks in the composed Petri nets. WorkFlow net with Data (WFD-net) was introduced in [53] as an extension of WF-Net in which a transition can have a guard, and can read from, write to, or delete data elements. Based on this net, data-flow anti-patterns comprising missing data, redundant data, lost data, conflicting data, never destroyed, twice destroyed, not deleted on time, were defined in terms of temporal logic CTL.

Differently from the previous work, we exploit the modeling power of CP-nets to provide a method for representing many aspects of a workflow, including data flow, control structure, and execution time, in one single model. Using CP-nets, we can avoid extending existing languages to represent the data factor, unlike the previous works. Data and changes on the properties of data can be represented explicitly too. We also show how to analyze the generated model to detect data abnormalities concerning the mutual influences among the data-related workflows instead of the interactions among the activities in a single workflow only. The method we use to model CSWs can be applied to model business workflows effectively.

5.3.3 Background

This section introduces the basic concepts of Colored Petri Nets [41, 54] that are used for modeling the essential behaviors of CSE in Sec. 5.3.4.

Definition 5.3.3.1. ***(Petri Net)*** *A PN is a triple* (P, T, A)*:*

- P *is a finite set of places.*
- T *is a finite set of transitions such that* $P \cap T = \emptyset$*.*
- $A \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation).*

Definition 5.3.3.2. ***(Multi-set)*** *Let* $S = \{s_1, s_2, s_3, ...\}$ *be a non-empty set. A* ***multi-set over S*** *is a function* $m : S \to N$ *that maps each element* $s \in S$ *into a non-negative integer* $m(s) \in N$ *called the number of appearances (coefficient) of* s *in* m*. The set of all multisets over* S*, i.e. the multiset type over* S*, is denoted* S_{MS}*.*

Definition 5.3.3.3. ***(Colored Petri Nets)*** *A* ***CP-net*** *is a tuple (P, T, A,* Σ*, N, C, G, E, I), where*

1. P *is a finite set of* ***places****.*
2. T *is a finite set of* ***transitions*** *such that* $P \cap T = \emptyset$*.*
3. $A \subseteq (P \times T) \cup (T \times P)$ *is a set of directed* ***arcs****.*
4. Σ *is a finite set of non-empty types, called* ***color sets****.*
5. V *is a finite set of* ***typed variables*** *such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$*.*
6. $C : P \to \Sigma$ *is a* ***color set function*** *that assigns a color set to each place.*
7. $G : T \to EXPR_V$ *is a* ***guard function*** *that assigns a guard to each transition* t *such that* $Type[G(t)] = Bool$*.*
8. $E : A \to EXPR_V$ *is an* ***arc expression function*** *that assigns an arc expression to each arc* a *such that* $Type[E(a)] = C(p)_{MS}$*, where* p *is the place connected to the arc* a*.*
9. $I : P \to EXPR_\emptyset$ *is an* ***initialization function*** *that assigns an initialization expression to each place* p *such that* $Type[I(p)] = C(p)_{MS}$*.*

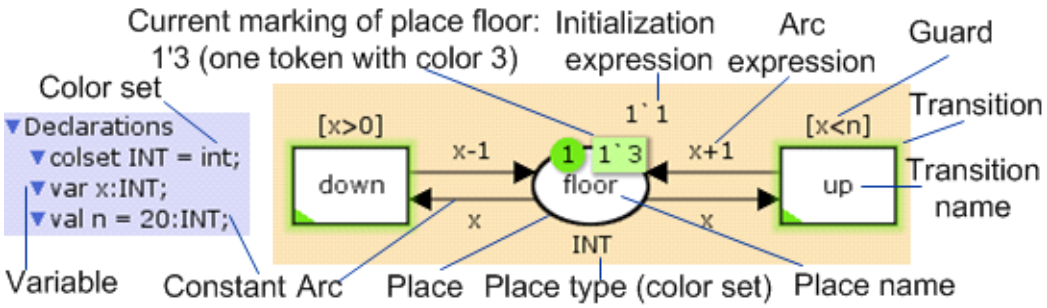


Figure 5.6: A CP-net modeling the behaviors of a simple elevator

Definition 5.3.3.4. *For a Colored Petri Net CPN = (P, T, A, Σ, N, C, G, E, I), we have the following concepts:*

1. *A **marking** is a function M that maps each place $p \in P$ into a multiset of tokens $M(p) \in C(p)_{MS}$.*
2. *The **initial marking** M_0 is defined by $M_0(p) = I(p)()$ for all $p \in P$.*

Definition 5.3.3.5. ***(Binding)** Let t be a transition of a CP-net.*

- *A **binding of transition** t allocates a concrete value to the variables that occur in the arc expressions of arcs surrounding t. These values should be of the corresponding type.*
- *Transition t is **enabled at a binding** if there are tokens that matching the values of the arc expressions and the guard of t evaluates to true.*
- *An enable transition **fires** while consuming and producing the corresponding tokens*

Fig. 5.6 shows an example of using CP-nets to model the behavior of a simple elevator. The elevator moves between twenty floors and can stop at any of these floors. At each floor, the elevator can go up or go down. We use the place *floor* with Integer type to model the floor. The token in this place has a color that denotes the floor number of the elevator. Because the elevator can travel up or down one floor, we model these events as transitions *up* and *down*. Transition *up* is enabled if the elevator is not at the top floor (i.e., the twentieth floor). Transition *down* is enabled if the elevator is not at the ground floor.

A CP-net can be extended with a *time concept*. A token in a CP-net has a concrete value, *color*, and in a timed CP-net, it may, in addition, carry a *time stamp* telling when it can be used (See the initial markings of the places $p11$, $p21$, and $p31$ in the subpages $csw1$, $csw2$, $csw3$ described in Fig. 5.12). Also, a transition may produce tokens with a *delay* (See Fig. 5.9a). To model tokens carrying a time stamp, the corresponding color set must be made a *timed color set* by adding the term *timed* (See the declaration of color set CSW in Fig. 5.7).

In addition to the *time concept*, a CP-net is also extended with a *hierarchy concept*. The idea is to decompose a CP-net into modules, *CP-net modules*. A module has an interface, *port place*, and is replaced by a *substitution transition*. A module is also referred to as a *page*. A substitution transition refers to a *subpage* that corresponds to a module

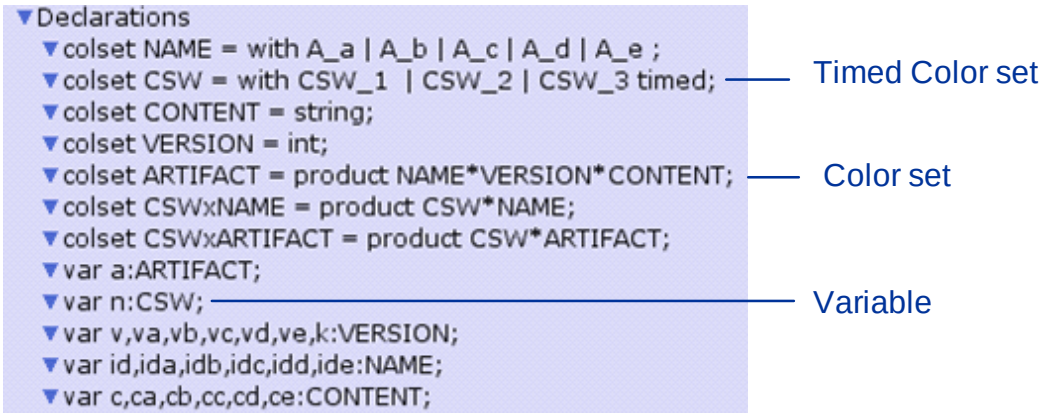


Figure 5.7: Color sets and variables used in CPN Model of CSE

contained in another module, named *superpage*. To connect a subpage with its superpage, each port place in the subpage is linked with a place in the superpage, named *socket*. By relating a socket to a port, the two places are semantically merged into one place. The basic concepts of a hierarchy CP-net are illustrated in Fig. 5.11. Following is the formal definition of a CP-net module.

Definition 5.3.3.6. ***(CP-net Module)*** *A CP-net Module is a four-tuple* $(CPN, T_{sub}, P_{port}, PT)$*, where*

1. CPN *=(P, T, A,* Σ*, N, C, G, E, I) is a* ***non-hierarchical CP-net****.*
2. $T_{sub} \subseteq T$ *is a set of* ***substitution transitions****.*
3. $P_{port} \subseteq P$ *is a set of* ***port places****.*
4. $P_{port} \rightarrow \{IN, OUT, I/O\}$ *is a* ***port type function*** *that assigns a* ***port type*** *to each port place.*

5.3.4 CPN Model of CSE

Color Set

Fig. 5.7 summarizes the color sets and variables used in modeling CSE by CP-nets. We represent the sets of artifacts and CSWs as color sets *ARTIFACT*, and *CSW* respectively. Other color sets are composed from the basic ones.

- The color set *ARTIFACT* is a product type representing the artifacts in CSE. The first element of the color set *ARTIFACT*, *NAME*, is an enumeration type that enumerates the artifact names. The second element, *VERSION*, is an integer type that denotes the version of an artifact. The third element, *CONTENT*, is a string type that denotes the content of an artifact. Value $(A\_b, 1, "b1")$ is an example of a color belonging to the color set *ARTIFACT* in which *A_b*, 1, and "*b*1" are colors of the color sets *NAME*, *VERSION*, and *CONTENT*, respectively.
- The color set *CSW* is an enumeration type that enumerates the identifications of CSWs. To represent the execution times E of the change activities, *CSW* is declared as a timed color set by adding the term *timed* at the end of its declaration.

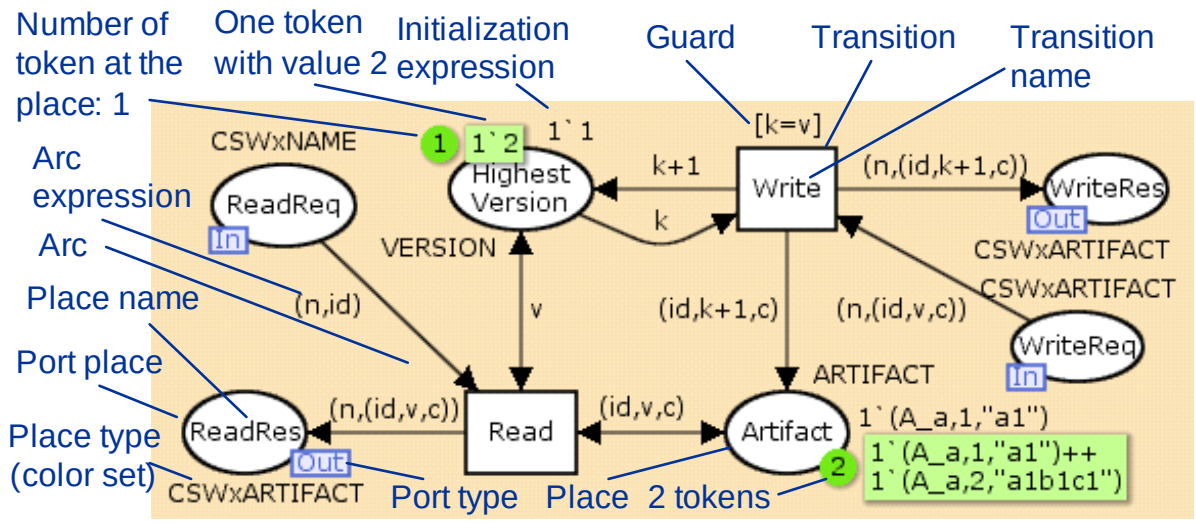


Figure 5.8: Modeling an artifact a as a simple VCS

Modeling Artifacts

In software development environments, VCSs are fundamental tools for enabling workers to work together in parallel on a single data repository. Based on this observation, we model each artifact as a simplified VCS to allow CSWs to access the artifacts concurrently (see Fig. 5.8). Each artifact is represented by a CP-net module with four port places: *ReadReq*, *ReadRes*, *WriteReq*, and *WriteRes* that represent a read (check-out) request, read response, write (check-in) request, and write response, respectively. CSWs will connect to these port places to access the corresponding artifacts. The state of an artifact is modeled by a place *Artifact*. This place contains tokens that represent all versions of the artifact created during the execution of CSWs involved. Tokens in the place *Artifact* will have the same name, but different version number. We assume that all CSWs will access the latest version of the artifact. Therefore, we use another place, *Highest Version*, containing only one token, to denote the highest version number of the artifact. When there is a read request, the transition *Read* will return the version that has the highest version number. When there is a write request, the transition *Write* will update the highest version number, and store and return the new version with the updated version number.

Fig. 5.8 shows a *marking* of the CP-net module modeling the artifact a. A *marking* represents a state of a CP-net and is determined by the number of tokens present in each place. In this marking, the place *Artifact* contains two tokens: one token with color *(A_a,1,"a1")*, denoted by *1‘(A_a,1,"a1")*, and one token with color *(A_a,2,"a1b1c1")*, denoted by *1‘(A_a,2,"a1b1c1")*. Therefore, the place *Highest Version* will contain one token with value 2, denoted by *1‘2*, which indicates that the latest version of the artifact a has the version number 2. The markings of the places *Artifact* and *Highest Version* are different from their *initial markings* which are represented by the corresponding initial expressions. In the initial marking, the place *Artifact* contains only one token with value *(A_a,1,"a1")*, *1‘(A_a,1,"a1")*, hence the token in the place *Highest Version* has value 1, *1‘1*.

To model other artifacts, for example the artifact b having only one version in which the

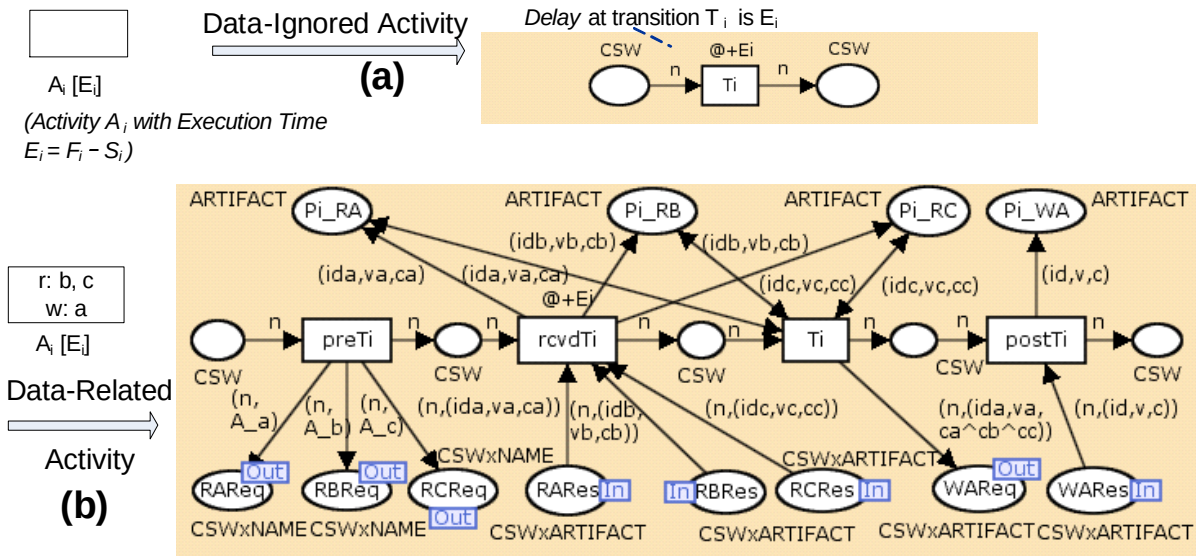


Figure 5.9: Modeling a change activity by CP-nets

version number is 1 and the content is "b1", we just need to change the initial expressions of the places *Artifact* and *Highest Version* to $1`(A\_b, 1, "b1")$ and $1`1$ respectively (See **Subpage:Accessb** in Fig. 5.11). If we want to represent an artifact that has not been created yet, we set the initial marking of the place *Highest Version* to *0* and do not set the initial marking for the place *Artifact*.

Modeling CSWs

Ignoring the data factor, each activity A_i can be modeled by a transition T_i. The execution time of A_i, $E_i = F(A_i) - S(A_i)$, is modeled by the delay time of transition T_i: $@+E_i$ (Fig. 5.9a).

In the case of data-related activities (Fig. 5.9b), each activity A_i in a CSW is modeled by four transitions: $preT_i$ (sending read requests), $rcvdT_i$ (receiving read responses), T_i (changing data and sending write requests), and $postT_i$ (receiving write responses). First, the transition $preT_i$ sends read requests by specifying the names of the data needed to be read, including the data in the read data set and the data in the write data set (except for the data not yet created), as the inscriptions of the arcs connecting $preT_i$ to the port places *ReadReq* of the read artifacts. Next, the transition $rcvdT_i$ receives the tokens returned from the port places *ReadRes* and stores them as local artifacts in some specific places, for example the place P_i_RA. P_i_RA means a place used for modeling an activity A_i, P_i, and containing the *read* value of an artifact a, RA. Then, the transition T_i sends tokens representing the updated versions of the changed artifacts to the port places $WriteReq$ of the corresponding artifacts. In this paper, we do not focus on the contents of artifacts, and we assume that the content of a written artifact is a concatenation of the contents of its read artifacts. Finally, the transition $postT_i$ receives tokens from the port places $WriteRes$ and stores them as local written artifacts in some specific places, for example the place P_i_WA. P_i_WA means a place used for modeling an activity A_i, P_i, and containing the *write* value of an artifact a, WA. The identification of the CSW containing the activity is also included in read/write requests and read/write responses

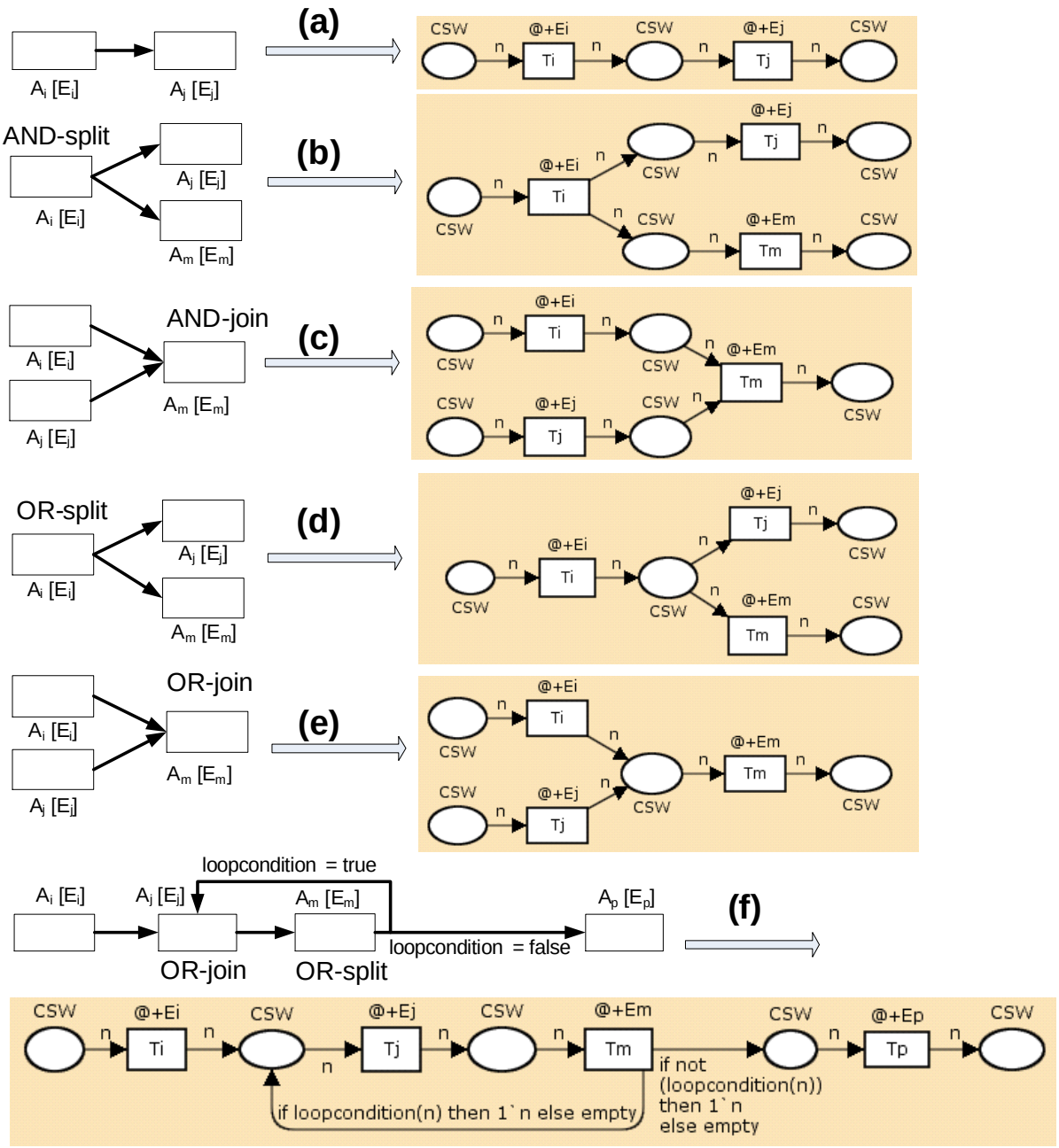


Figure 5.10: Modeling basic constructions of a CSW by CP-nets

to ensure responses are delivered correctly. The execution time E_i of A_i is modeled by the delay time of the transition $rcvdT_i$: $@+E_i$.

Fig. 5.10 shows how the basic constructions of a CSW are modeled by CP-nets: sequential structure (Fig. 5.10a), parallel structure with an AND-split activity (Fig. 5.10b), parallel structure with an AND-join activity (5.10c), selective structure with an OR-split activity (Fig. 5.10d), selective structure with an OR-join activity (Fig. 5.10e), and iterative structure (Fig. 5.10f). Based on these structures, we can develop more complicated structures using inscriptions and expressions supplied by CP-nets to specify the conditions of route choices, type and quantity of tokens (resources, data) transmitted,

and execution condition of transitions, etc. For simplicity, we only represent the control flow and time factor in these structures. Therefore, when representing a CSW in CP-nets, one should model the control flow first using the above structures. Then, one can model the data factor of each data-related activity by replacing its corresponding transition by the four transitions as described in Fig. 5.9b.

Fig. 5.11 and Fig. 5.12 give an example of modeling a CSE with three CSWs, $CSW1$, $CSW2$, $CSW3$, connected by five artifacts, a, b, c, d, e, in terms of CP-net. In this CSE, $CSW2$ and $CSW3$ represent the CSWs of Tom and Mary presented in Motivating Example of Sec. 2.2. The artifacts d, e, b represent the *showPoint()* method of the $VIPCustomer$ class, the *showPoint()* method of the *RegularCustomer* class, and the *showCustomerScreen()* method of the *Display* class, respectively.

The hierarchical structure is used in this example. The superpage contains eight subpages: $csw1$, $csw2$, $csw3$, *Accessa*, *Accessb*, *Accessc*, *Accessd*, and *Accesse* that represent $CSW1$, $CSW2$, $CSW3$, a, b, c, d, and e, respectively. As described in Sec. 5.3.3, the port places of the subpages will be linked to the sockets of their superpages. In this example, the sockets *RCReq*, *RCRes*, *WCReq*, and *WCRes* of the superpage will link to the port places *ReadReq*, *ReadRes*, *WriteReq*, and *WriteRes* of the subpage *AccessC*, respectively. And so are the remaining sockets.

5.3.5 Detecting Abnormalities in CPN Model of CSE

In the CPN model of CSE, data-related CSWs are connected and create a *synthesized CSW-net* (See the **Superpage** in Fig. 5.11). We can simulate and analyze the synthesized CSW-net using the simulation tools, and state space tools with state space querying functions, CTL (Computation Tree Logic) state formula operators, and model checking functions, supplied by CPN Tools [42], to detect abnormalities.

Missing Data and Direct Conflict

Missing data is the situation in which an artifact needs to be read, but either it has never been created or it has been deleted. In CSE, missing data happens on an artifact a if the place *Artifact* corresponding with it contains no token. Therefore, there is a deadlock at a transition $rcvdT_i$ that is waiting for a read response from a.

A direct conflict happens on an artifact a if there is a deadlock at a transition $postT_i$ that is waiting for a write response from a. This deadlock happens because a write response is returned only if the version number in the write request is equal to the number stored in the place *Highest Version* corresponding to the artifact a (See the guard of the transition *Write* in Fig. 5.8 or Fig. 5.11). This condition simulates the check-in condition of VCSs. Our modeling method can detect conflicts caused by not only concurrent activities in different CSWs (**Inter-Direct-Conflict**) but also concurrent activities in the same CSW (**Intra-Direct-Conflict**).

Using the simulation tools or state space analysis report of CPN Tools, we can detect **missing data** and **direct conflict** easily. For instance, Fig. 5.13 shows the standard report generated for the state space analysis of the CPN model described in Fig. 5.11 and Fig. 5.12. We observe the absence of the home markings. There are no infinite occurrence sequences and the CSW-net terminates at the dead marking 49. The $Accessc'Write$ transition is dead because there is no write request for the artifact c. Standard reports can

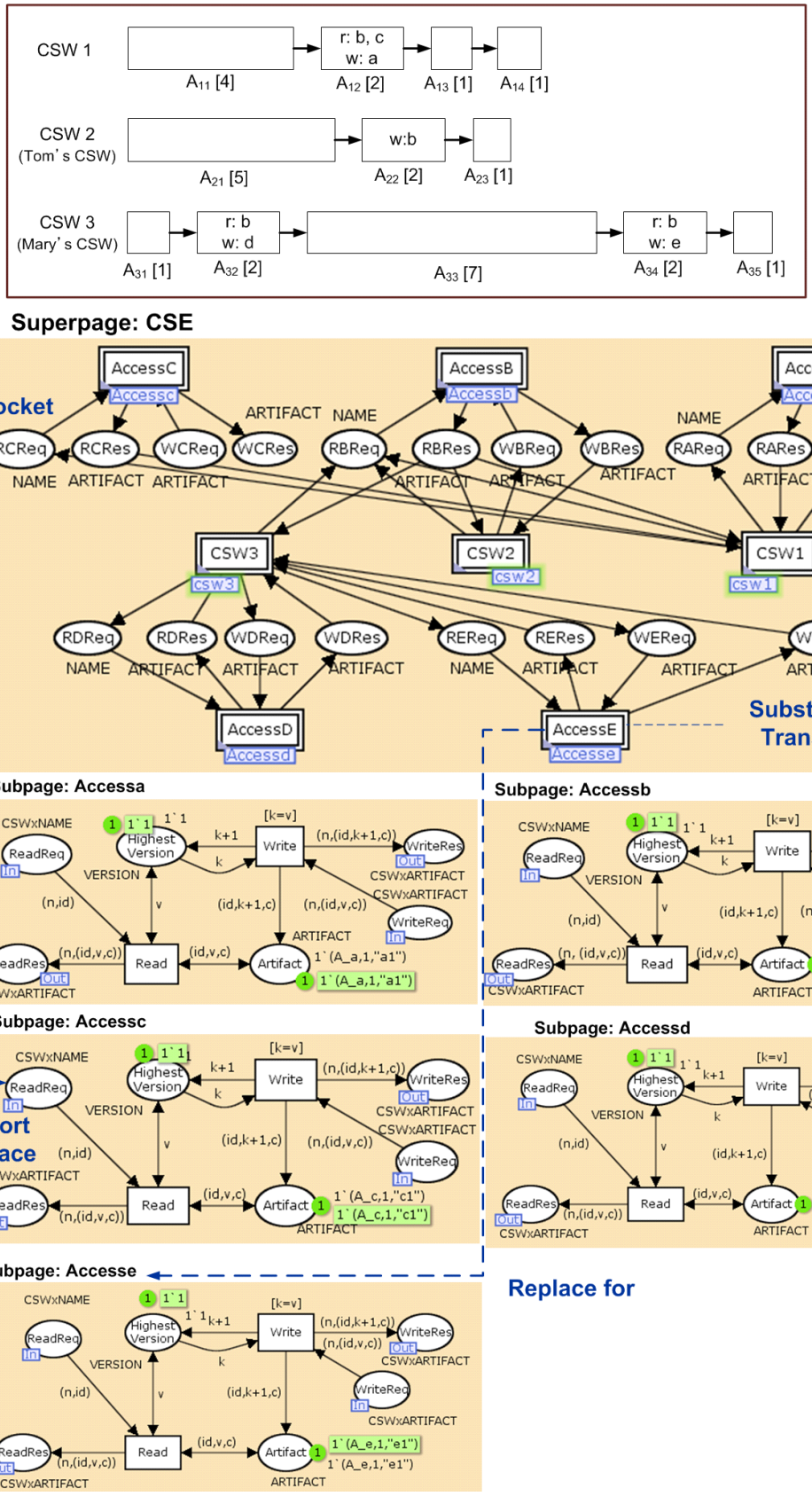
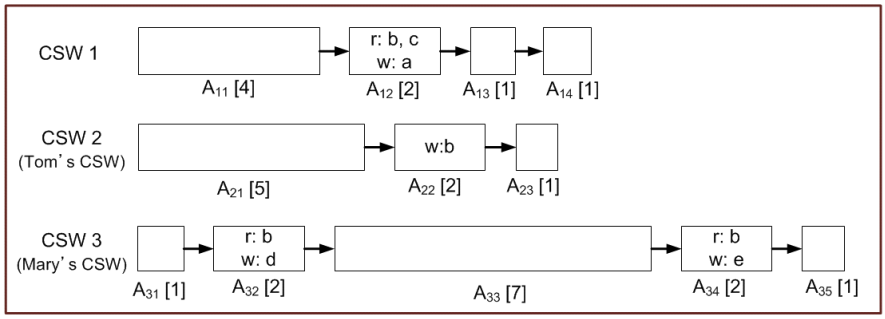
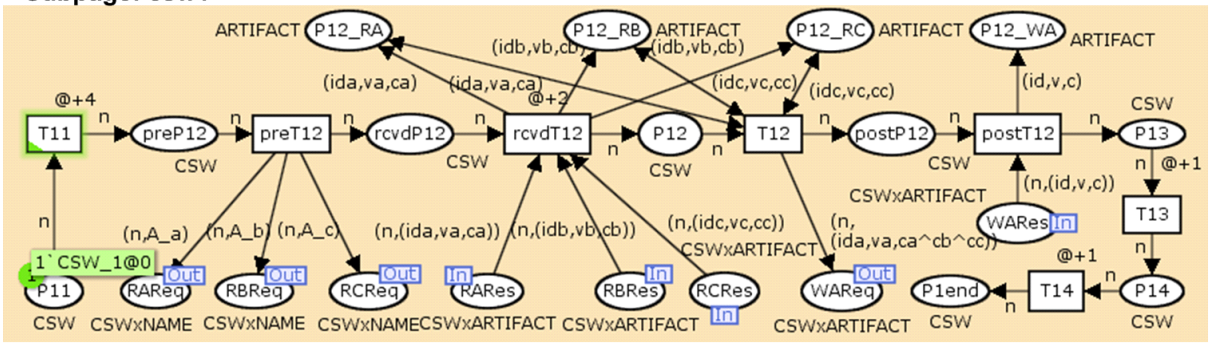


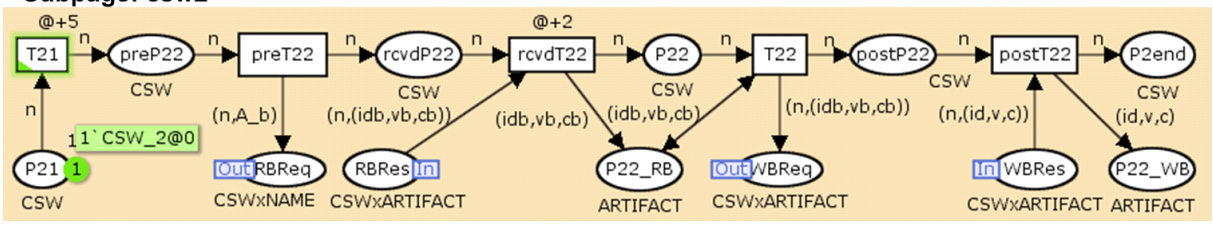
Figure 5.11: An example of modeling and verifying a CSE with 3 CSWs and 5 artifacts by CP-nets (Part 1)



Subpage: csw1



Subpage: csw2



Subpage: csw3

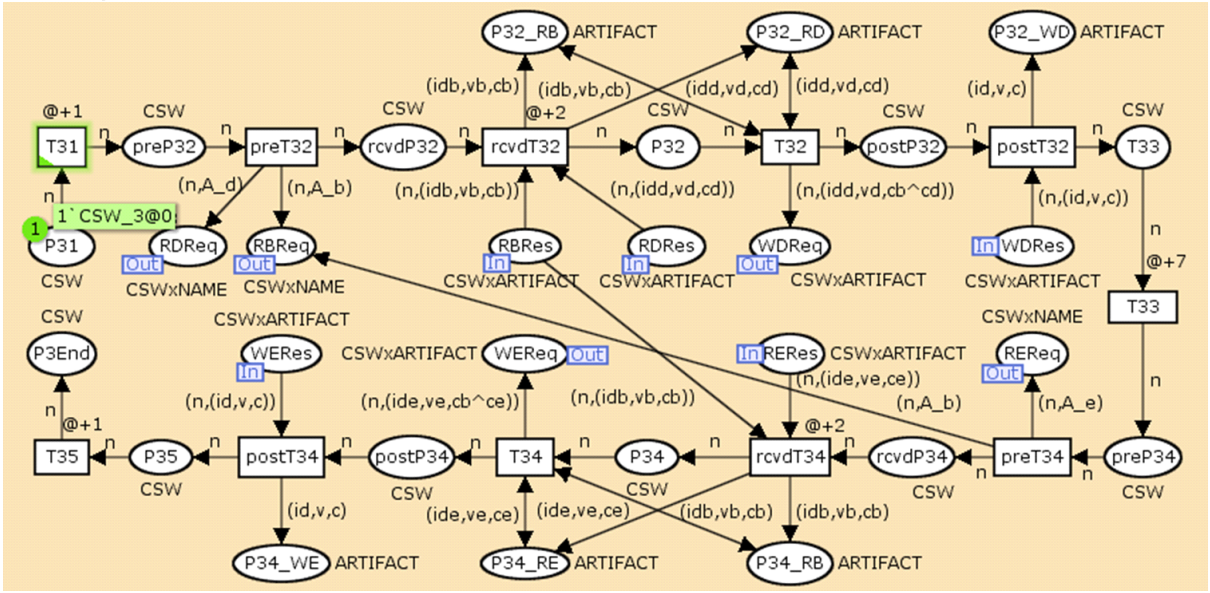


Figure 5.12: An example of modeling and verifying a CSE with 3 CSWs and 5 artifacts by CP-nets (Part 2)

```
Statistics
------------------------------------------------------------
  State Space
      Nodes:  49
      Arcs:   63
      Secs:   0
      Status: Full
  Scc Graph
      Nodes:  49
      Arcs:   63
      Secs:   0

Boundedness Properties
------------------------------------------------------------
  Best Integer Bounds
                                  Upper       Lower
      Accessa'Artifact 1          2           1
      Accessa'Highest_Version 1   1           1
      Accessb'Artifact 1          2           1
      Accessb'Highest_Version 1   1           1
      Accessc'Artifact 1          1           1
      Accessc'Highest_Version 1   1           1
      ..........................

  Home Properties
------------------------------------------------------------
  Home Markings: Initial Marking is not a home marking

Liveness Properties
------------------------------------------------------------
  Dead Markings: [49]
  Dead Transition Instances: Accessc'Write 1
  Live Transition Instances:  None

Fairness Properties
------------------------------------------------------------
      No infinite occurrence sequences.
```

Figure 5.13: The standard report generated for the state space analysis of the CPN model described in Fig. 5.11 and Fig. 5.12

help us detect abnormalities easily, especially by observing dead markings, or boundedness properties. In this case, no abnormalities are detected through the standard report. This means that there are no errors relating to **missing data** or **direct conflicts**.

Potential Direct-Revision-Inconsistency

Potential Direct-Revision-Inconsistency can be recognized by observing the superpage of the CPN model of CSE. If there is more than one connection to a socket *WAReq* linking with a port place *WriteReq* of a subpage *Accessa* that models an artifact a, there may be a direct conflict or a potential Direct-Revision-Inconsistency related to the artifact a. If no deadlock happens, we can eliminate the direct conflict case. For example, in the Superpage of the CPN model described in Fig. 5.11, there is at most one connection to the sockets *WAReq, WBReq, WCReq, WDReq, WEReq*. Therefore, a **Direct-Revision-Inconsistency** does not happen.

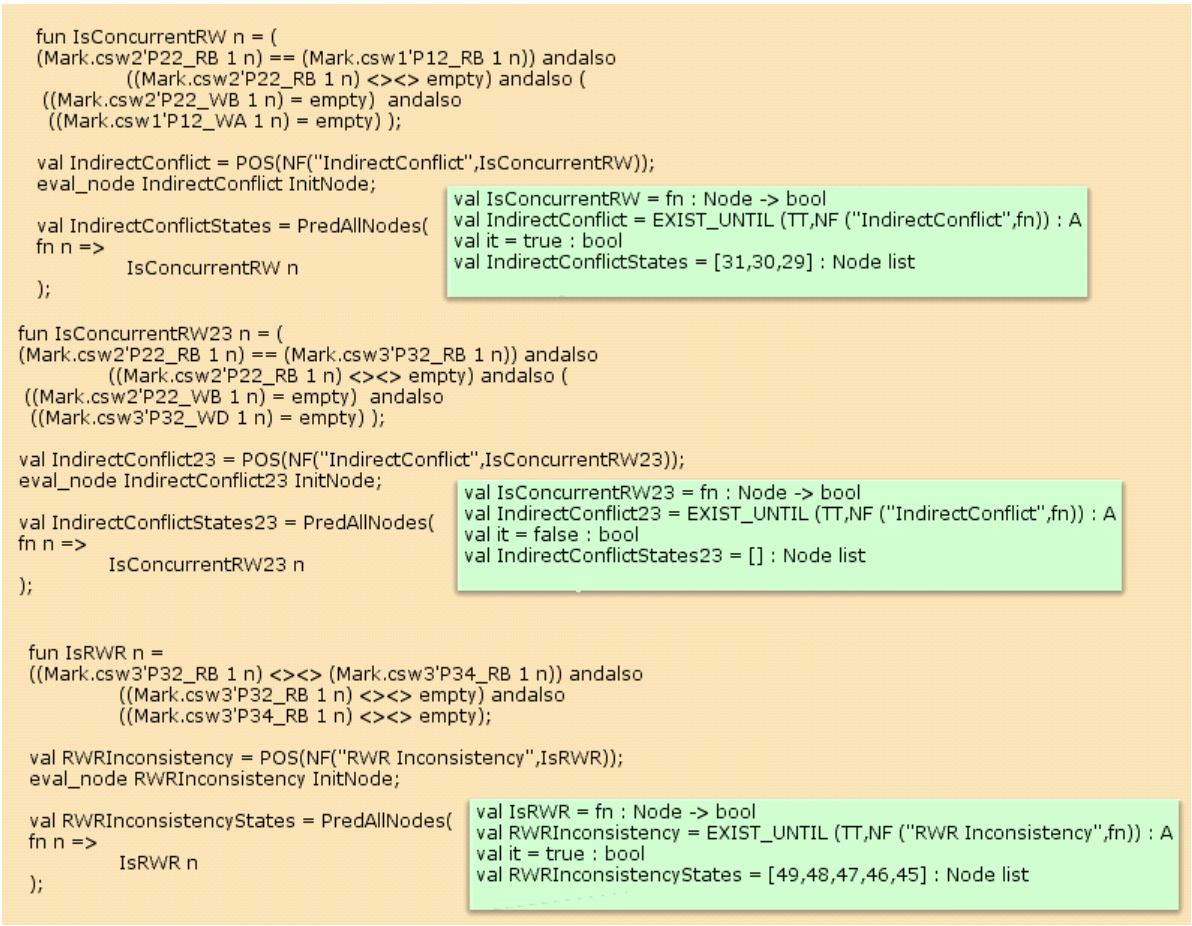


Figure 5.14: Example of model checking and query on the CPN model described in Fig. 5.11 and Fig. 5.12

Other Potential Inconsistencies

State space querying and model checking are necessary to detect the remaining patterns of inconsistency. We will show a concrete example of applying state space queries and CTL model checking functions on the CPN model in Fig. 5.11 and Fig. 5.11 to detect these patterns.

The Superpage in Fig. 5.11 reveals that the artifact b is read by $CSW1$ and $CSW3$, and read/written by $CSW2$. Hence, we will do some verifications with b.

As shown in Fig. 5.14, we first check the possibility of an **Indirect-Conflict**. The function *IsConcurrentRW* checks whether the artifact b is concurrently read and written by the activities A_{12} and A_{22} respectively. Because *IndirectConflict* is evaluated as *true*, we conclude that a potential Indirect-Conflict happens between A_{12} and A_{22} at some states. We can find these states by a query function, *IndirectConflictStates*, that traverses the entire state space to find the states satisfying the *IsConcurrentRW* function. Also, a potential Indirect-Conflict between A_{12} and A_{22} means that a potential **Indirect-Revision-Inconsistency** does not happen between these two activities.

Similarity to the function *IsConcurrentRW*, the function *IsConcurrentRW23* checks whether the activities A_{32} and A_{22} concurrently read and write the artifact b. Because *IndirectConflict23* is evaluated as *false*, we conclude that a potential **Indirect-Revision-**

Inconsistency happens between A_{32} and A_{22}.

Next, we check the possibility of a potential **RWR Interleaving-Inconsistency**. The function *IsRWR* checks whether the activities A_{32} and A_{34} read the same value of b. Similar to the previous case, RWR Interleaving-Inconsistency happens because of the *true* value of *RWRInconsistency*. *RWRIncosistencyStates* returns a list of states that make the function *IsRWR* true.

Although a potential **WWR Interleaving-Inconsistency** does not appear in this example, we can imitate the method checking potential RWR Interleaving-Inconsistency to detect this inconsistency. We will define a function *IsWWR* for checking whether an activity A_j in a CSW *csw* reads the value created before by an activity A_i in the same CSW. The remaining functions are defined similarly to those of RWR Interleaving-Inconsistency except that *RWR* is replaced with *WWR*.

fun isWWR n = ((Mark.csw'Pi_WB 1 n) <><> (Mark.csw'Pj_RB 1 n))
and also (Mark.csw'Pi_WB 1 n) <><> empty)
andalso ((Mark.csw'Pj_RB 1 n) <><> empty);

5.3.6 Discussion

We have presented a formal model of our proposed Change Support Environment (CSE) which represents the change processes as Change Support Workflows (CSWs) and manages their execution for dealing with inconsistencies in collaborative environments more effectively. CP-nets are used to model the necessary behaviors of CSE, in particular the data flows of CSWs. CPN Tools is then used to edit, simulate, and verify the CPN model of CSE to detect data abnormalities, specially the patterns of inconsistency.

Although our approach can successfully model and verify CSE, this modeling approach could not take into account all aspects of a real CSE to reduce the complexity of the generated model. Also, due to the modeling cost and the state explosion problem in model checking, this approach is suitable for small-size CSEs with CSWs that need to be designed carefully and to verify data-flow related errors in advance, before executing them. Despite the limitations, successful modeling and verification of CSE are the initial achievements in proving the feasibility and correctness of the approach of CSE. In addition, this modeling and verification method with some advantages compared to the previous works in data-flow verification can be applied to other types of workflows, such as business worfkflow.

In future work, we will continue improving the formal model of CSE with regard to modeling other common operations on artifacts such as branching and merging, and representing the indirect dependencies among artifacts. Automating the inconsistency analysis is also worth considering. Moreover, how to derive the correct CSWs from the inconsistency-involved CSWs is under consideration.

5.4 Summary

This chapter has gone into detail our inconsistency awareness technique that is a combination of the workspace awareness technique and context awareness technique. By analyzing the latest changes in the workspaces of the workers and the progress of the change processes in the system, we can notify the workers of a (potential) inconsistency in advance and its context, which supplies not only the contents of the changes involved but also the

change processes containing these changes, to help them comprehend the situation and have suitable decisions to resolve the inconsistency. The alternative method for modeling and verifying CSE to detect the patterns of inconsistency is also a promising solution useful for small size CSEs with important CSWs. Because our goal is to support collaborative software development in practice, we adopt the inconsistency awareness technique to develop an inconsistency management support system presented in the next chapter.

Chapter 6

An Inconsistency Management Support System for Collaborative Software Development

This chapter presents a Change Support Workflow Management System (CSWMS) developed based on the theoretical model of CSE and the inconsistency awareness technique presented in Section 5.2. A brief description of development and implementation of CSWMS is given.

6.1 Requirements

In CSWMS, each change process is represented by a CSW, and workers follow CSWs to implement their change requests.

In collaborative software developments, a change process is different in nature from business processes as follows:

- A change process structure is simpler and data-oriented.
- A change process could not be known entirely at the beginning, and will evolve during its execution, because a worker cannot estimate all the impacted elements of a change request until the change is implemented.

Because of the above differences and the need for inconsistency awareness support, there are some specific requirements for CSWMS:

- CSWs must be easy to define and execute. Workers can define CSWs from scratch or by cloning existing CSWs.
- CSWs can be modified during execution. Workers can add, remove, or change activities in their CSWs anytime.
- CSWMS supports the workers in detecting and resolving inconsistencies among the CSWs defined or executed within a given time interval, by notifying them in advance of a (potential) inconsistency and its context. Workers should examine the context of an inconsistency, including the contents of the related changes and their CSWs, to fully understand the situation and decide if the reported inconsistency will lead to a real inconsistency.

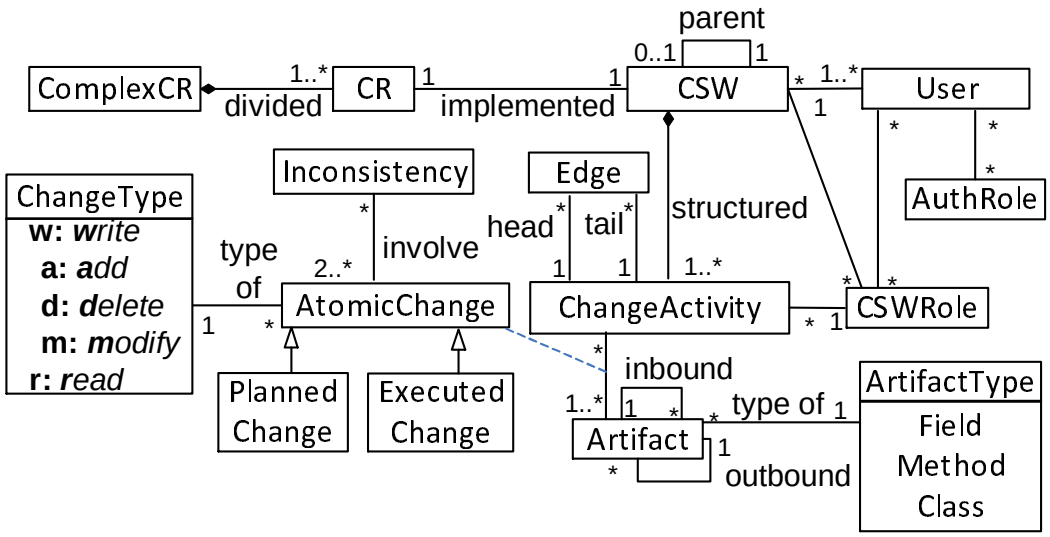


Figure 6.1: Static model of CSWMS

6.2 Static Model

Fig. 6.1 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* artifacts, and can also be depended by many artifacts, *inbound* artifacts. The type of an *Artifact* is specified by *ArtifactType* that can be a field, a method, or a class.
- A *ChangeActivity* can apply many changes, *AtomicChanges*, to an *Artifact*, and an *Artifact* can be changed by many *ChangeActivities*. We distinguish two types of *AtomicChange*: *ExecutedChange* is a change made at runtime, and *PlannedChange* is a change specified by a worker at build time. Each *AtomicChange* makes a *ChangeType* to an *Artifact*, which can be ***a**dd*, ***d**elete*, or ***m**odify*.
- An inconsistency, *Inconsistency*, involves at least two *AtomicChanges*.
- A *ChangeActivity* is executed by a CSWMS user, *User*. A *User* of CSWMS is associated with some authentication role, *AuthRole*, for example, admin or designer. Also, a *CSW* defines its specific roles, *CSWRoles*. A *User* can execute a *ChangeActivity* if a *CSWRole* is associated with both the *ChangeActivity* and the *User*.

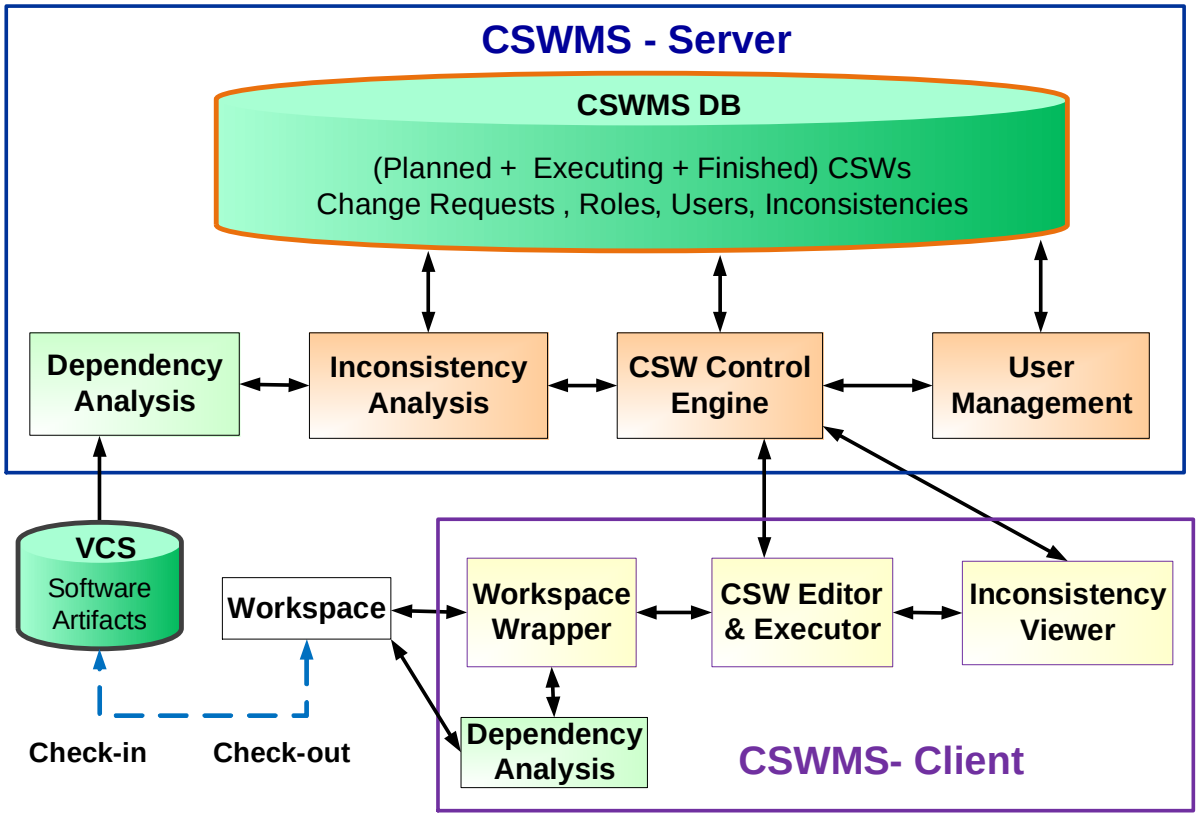


Figure 6.2: CSWMS architecture

6.3 Architecture

To realize the inconsistency awareness mechanism presented in Section 5.2.3, we develop the CSWMS with three main functions: CSW management, workspace monitoring, and inconsistency awareness. The first and second functions supply the necessary information to the third function to analyze inconsistencies. Fig. 6.2 shows the client-server architecture of CSWMS. The first function is implemented by the **CSW Editor & Executor**, **CSW Control Engine**, **CSWMS DB**, and **User Management** 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 that supports the **Workspace Wrapper** and **Inconsistency Analysis** components.

A VCS is used to manage the artifacts in the system. Workers will implement change requests in their workspaces.

CSWMS Server includes:

- **CSWMS DB:** contains information of the planned, executing, and finished CSWs in the system, Change Request, Roles, Users, and Inconsistencies.

  User Management: manages Users and Roles in the system. Level of detail in showing the states of CSWs in the system and (potential) inconsistencies will vary depending on the assigned roles of workers.

- **CSW Control Engine:** controls the execution of CSWs in the system.

- **Inconsistency Analysis:** notifies the workers of (potential) inconsistencies by analyzing the information received from clients, including the changes to the CSWs and ongoing changes in the workspaces of users, information stored in CSWMS DB, and information about the dependencies among the software artifacts provided by **Dependency Analysis**.

CSWMS Client includes:

- **CSW Editor & Executor:** allows a worker to edit a CSW. He can also execute a CSW and modify it during the execution. A worker can control the outgoing information by setting up the level of detail in broadcasting the CSW Progress that includes the changes in his workspace and the changes to his CSWs.
- **Inconsistency Viewer:** shows the contexts of the inconsistencies reported by the CSWMS server, including the changes causing the inconsistencies and their CSWs. A worker can set up the level of detail of the incoming information and register for favorite information depending on his roles in CSWMS.
- **Workspace Wrapper:** tracks activities in the workspaces of workers to extract the information necessary for inconsistency analysis, and sends some commands to the Workspace.

The **Dependency Analysis** component appears on both client and server. It helps **Workspace Wrapper** and **Inconsistency Analysis** analyze the dependencies among the software artifacts.

6.4 Dynamic Model

This section explains three main scenarios in a CSWMS: editing a CSW, executing a change activity, and inconsistency awareness.

6.4.1 Editing CSWs

CSWMS users use **CSW Editor & Executor** to define a CSW for implementing a change request. They can use impact analysis [36] tools to identify the impact of the change request with the start impact set as a *root artifact* (1, 1.1, 1.2). Because the result of impact analysis is just potential changes, one should revise the returned set to increase the precision of their CSWs. However, impact analysis techniques are outside the scope of this dissertation.

CSWMS users can still modify their CSWs at runtime, such as by adding or removing a change activity and the corresponding artifacts, reassigning workers for executing activities, and modifying the orders of the activities. All the changes to CSWs of the users will be sent to **CSW Control Engine**, and **CSW Editor & Executor** will operate depending on CSW Progress Response. (2, 2.1, 2.2)

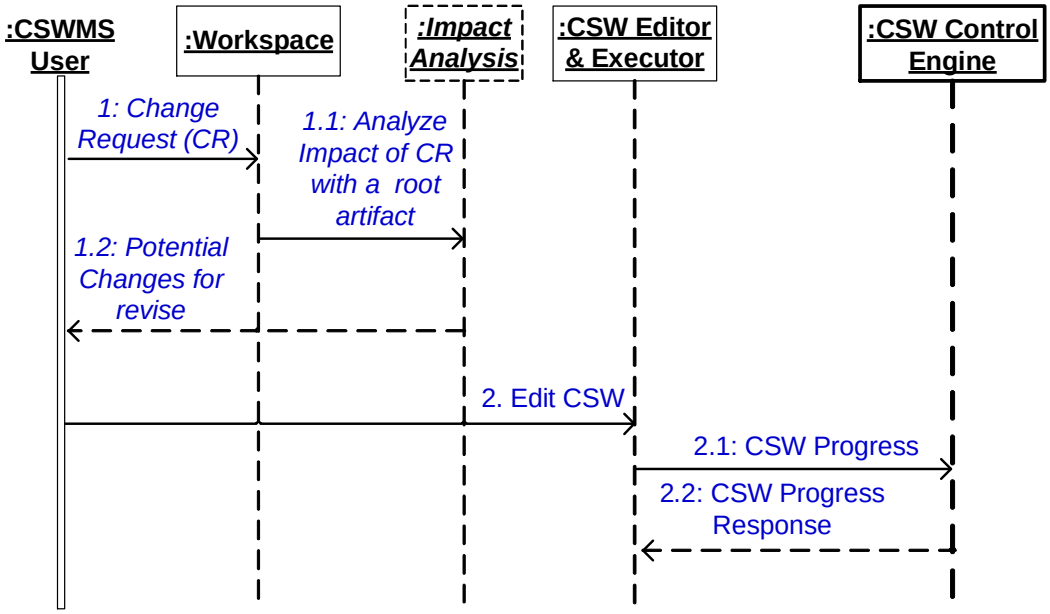


Figure 6.3: CSW editing scenario

6.4.2 Executing a Change Activity

When a CSWMS user starts a change activity (3), his workspace will be synchronized with the remote repository (3.3, 3.3.1, 3.3.2). **Workspace Wrapper** monitors the workspace of the user to obtain the ongoing changes (4, 4.1, 4.1.1, 4.1.2), and reports the atomic changes to **CSW Editor & Executor** (4.2). When the user finishes the change activity (5), the changed artifacts will be checked-in if no inconsistency is detected (5.1, 5.2, 5.3, 5.4, 5.5, 5.5.1, 5.5.2). During activity execution, **CSW Editor & Executor** will send CSW Progress, including the atomic changes and the changes to CSW, to **CSW Control Engine**, and have suitable actions depending on CSW Progress Response returned from **CSW Control Engine** (3.1 and 3.2, 4.3 and 4.4, 5.3 and 5.4).

6.4.3 Inconsistency Awareness

Inconsistency awareness of CSWMS is a combination of workspace awareness (recognizing the ongoing changes in the workspaces) and context awareness (recognizing the changes to CSWs). Based on the patterns of inconsistency, **Inconsistency Analysis** detects (potential) inconsistencies (6.1, 6.1.1 and 6.1.2, 6.1.1a and 6.1.1a.1, 6.2) by analyzing CSW Progress (6), including the ongoing changes in the workspaces and the changes to CSWs, received from the clients, the dependency relationships among the artifacts (6.1.1, 6.1.2), and the involved activities of the (planned, executing or finished) CSWs defined or executed within a given time interval (6.1.1a, 6.1.1a.1).

Inconsistencies will be sent to the users involved (6.3a, 6.3a.1). Upon receiving a warning of (potential) inconsistencies from CSWMS, one should examine the context of the reported inconsistency to solve it or ignore it in case of a false warning.

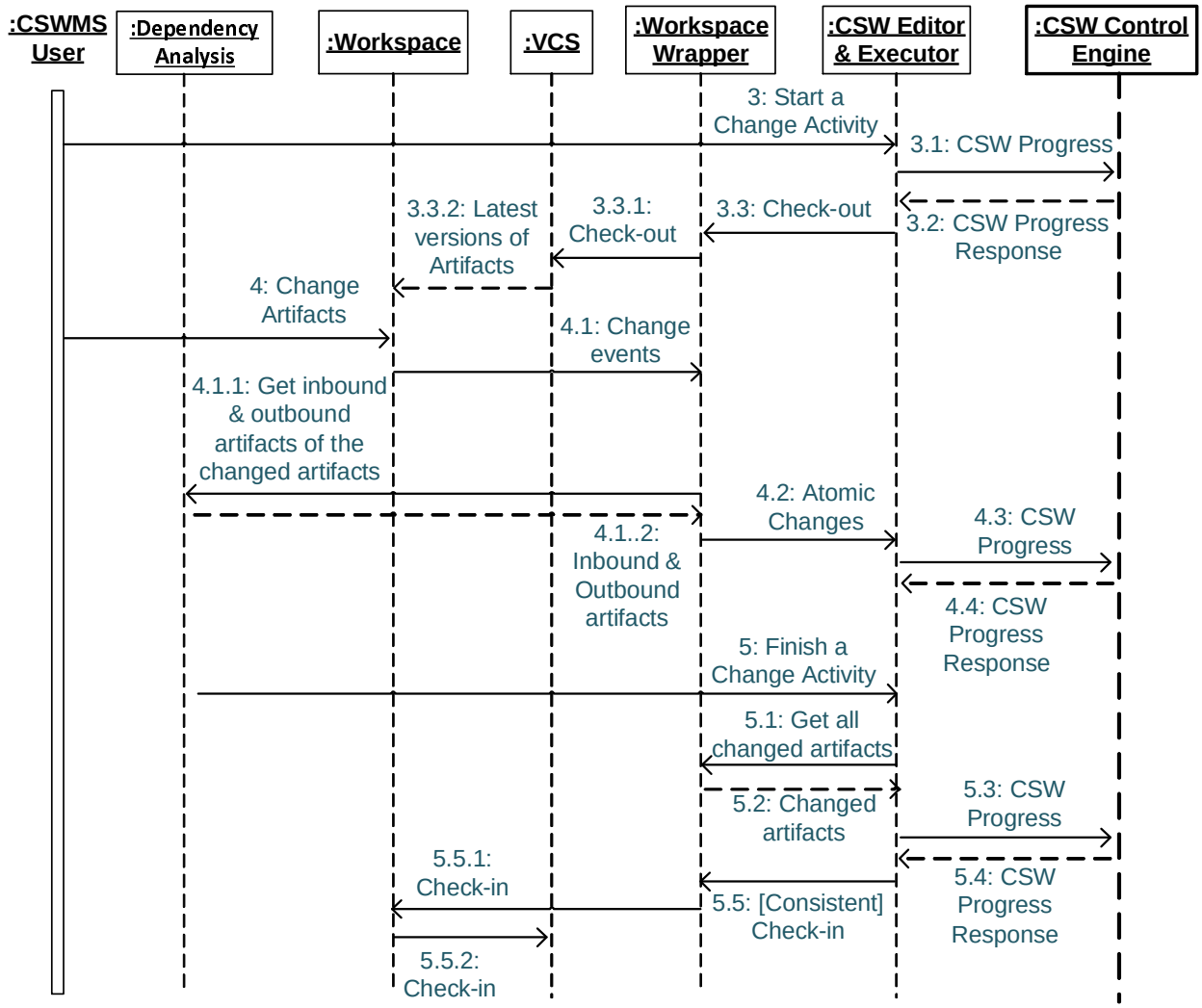


Figure 6.4: Change activity execution scenario

6.5 CSWMS - Implementation

Bonita 3.1 [33] is a WFMS that supports *collaborative process*, a workflow process allowing definition and execution operations just after the process is created, which satisfies the above requirements of CSWs for being modified and executed easily and dynamically. Therefore, we do not develop CSWMS from scratch, but extend and customize this open source package with functions relating to artifact management and inconsistency awareness. Fig. 6.6 shows the technical architecture of the CSWMS prototype.

6.5.1 CSW Management

CSW Editor & Executor allows a CSWMS user to edit and execute a CSW, or modify it during its execution. We develop it as a standalone application by customizing the Activity Manager and Workflow Graph Editor of Bonita 3.1 Client [33], and using the open source JGraph 5.14.0 [34].

CSW Control Engine, **User Management**, and **CSWMS DB**, common compo-

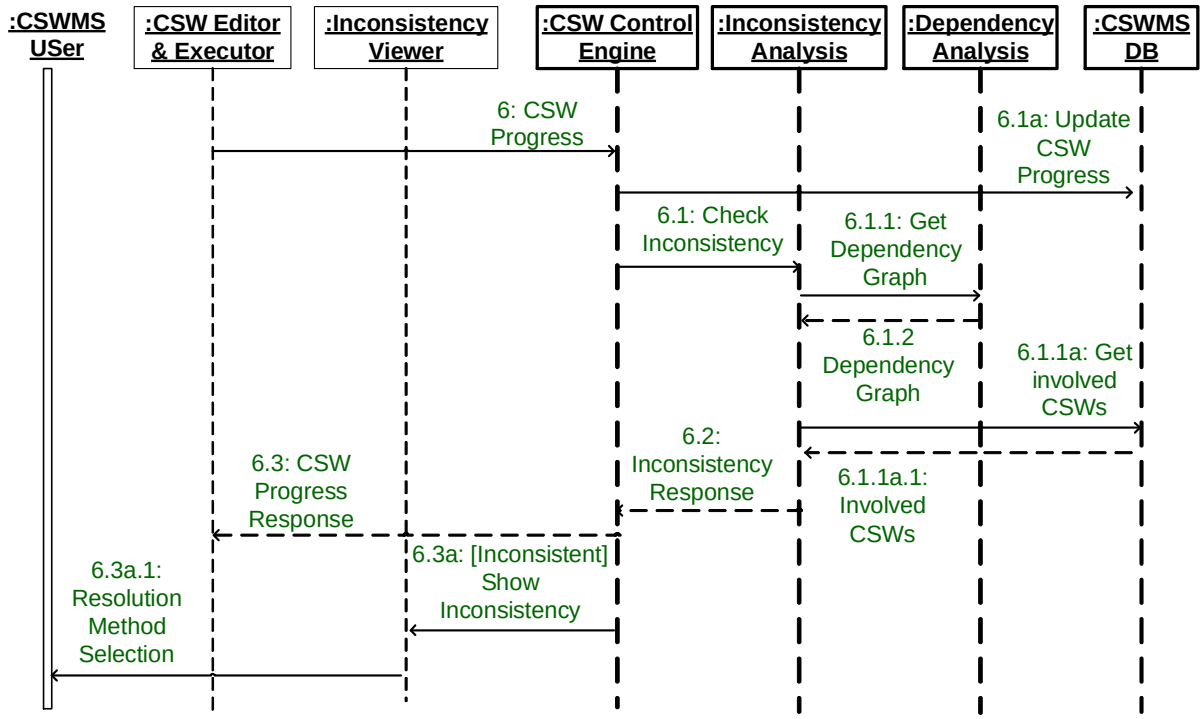


Figure 6.5: Inconsistency awareness scenario

nents of a WFMS, are developed by customizing and extending Bonita Engine 3.1 [33] to fit the requirements of CSWMS. For example, new data, such as atomic changes and inconsistencies, are added to **CSWMS DB**.

6.5.2 Workspace Monitoring

The **Workspace Wrapper** component monitors the workspaces of the CSWMS users to find out about the atomic changes that are being made in the workspaces by the users, and then send them to server for inconsistency analysis. In the current prototype, this component is implemented as Eclipse plugins to automatically collect the atomic changes to Java source codes on Eclipse workspaces of the users, *Executed Changes* (Fig. 6.7-E). Using Eclipse JDT (Java Development Tools), Eclipse AST (Abstract Syntax Tree), and Resource Change Tracking techniques (org.eclipse.core.resources.IResource-ChangeListener) of Eclipse, we can identify which type of change (add, delete, modify, or Subversion Synchronize) applied to a Java element, the *main artifact* of an atomic change. **Dependency Analysis** is then used to identify the other elements of an atomic change including the *inbound artifacts*, the Java elements that depend on the *main artifact*, and the *outbound artifacts*, the Java elements on which the *main artifact* depends (Fig. 5.3). To increase the accuracy of inconsistency analysis, we capture the changes at the client workspaces at the level of program entities. Currently, the type of an artifact in CSWMS can be a field, a method, or a class.

The reported atomic changes vary depending on the type of change, *add*, *delete*, or *modify*, and which Java elements, *class*, *method*, or *field*, are affected.

- If a field is added to a class, report two atomic changes that are the addition of the

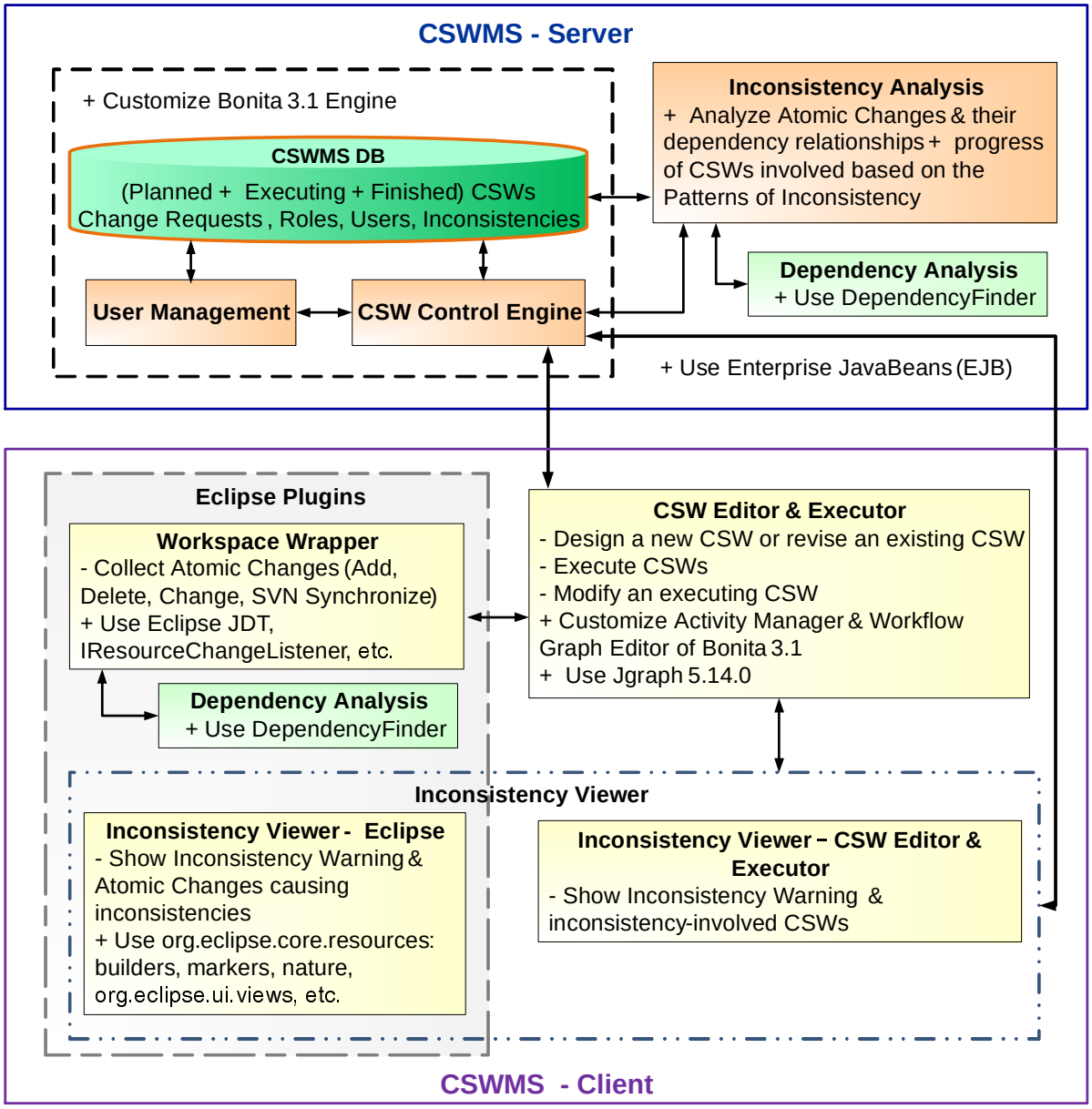


Figure 6.6: Technical architecture of the CSWMS prototype

new field and the modification of the containing class.

- If a field is removed from a class, report two atomic changes that are the deletion of the field and the modification of the containing class.
- If a field is modified, report two atomic changes that are the modification of the field and the modification of the containing class.
- If a method is added to a class, report two atomic changes that are the addition of the new method and the modification of the containing class.
- If a method is removed from a class, report two atomic changes that are the deletion of the method and the modification of the containing class.

- If a method is modified, report two atomic changes that are the modification of the method and the modification of the containing class.
- If a class is added or deleted, report an atomic change involving with the addition or deletion of the class.
- Besides the addition of a new field|method and the deletion of a field|method, if there is a change on other properties of a class, such as adding or removing an import statement, or modifying the access modifier of the class, report an atomic change involving with the modification of the class.
- If a class|method|field is renamed, it is considered as the deletion of the class|method|field with the old name and the addition of the class|method|field with the new name.

6.5.3 Inconsistency Awareness

The **Inconsistency Analysis** component analyzes the atomic changes received from the clients and the progress of CSWs involved by implementing the mechanism presented in Section 5.2.3. Algorithm 1 shows the pseudocode of the inconsistency detection algorithm of CSWMS. This algorithm is triggered by the arrival of a new atomic change C sent from a CSWMS client. It then calls the *AnalyzeInconsistencyonAtomicChangeswithSameArtifact* function, which finds the atomic changes C' stored in **CSWMS DB** having the same main artifact with C, and the *AnalyzeInconsistencyonAtomicChangeswithDependencyRelatedArtifacts* function, which finds the atomic changes C' having the main artifact depending on or depended by the main artifact of C. Further analyses are performed to identify the potential inconsistencies based on the necessary conditions of the inconsistency. The first function detects Intra|Inter-Direct-Conflict, potential Direct-Revision-Inconsistency, and potential WWR Interleaving-Inconsistency in Plan. The second function detects potential Indirect-Conflict, Indirect-Revision-Inconsistency, RWR Interleaving-Inconsistency in Plan, and RWR|WWR Interleaving-Inconsistency. Finally, the algorithm returns a list of potential inconsistencies. In the current prototype, we concentrate on analyzing inconsistency relating with the changes to methods or fields of a class to increase the possibility that the detected situations will lead to real inconsistencies.

The **Inconsistency Viewer** component 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. 6.7-F1,G1). Inconsistency Viewer plugins show the inconsistency warnings directly in Eclipse IDE (Fig. 6.7-F2,F3), together with the contents of the changes (source codes) causing the inconsistencies (Fig. 6.7-G2), using Eclipse Builders, Markers, Nature, Views, etc.

6.5.4 Dependency Analysis

The **Dependency Analysis** component appears on both client and server. It uses DependencyFinder [35], a third party tool for creating a dependency graph from compiled Java code, to analyze the dependencies among the software artifacts. 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

Algorithm 1: Detect Potential Inconsistencies

input : C.artifact (the main artifact of C),C.inbounds (the inbound artifact list of C), C.outbounds (the outbound artifact list of C), C.activity (the change activity containing C), C.csw (the CSW containing the activity), C.project (the Java project containing the artifacts), C.author (the worker making the change C), and C.version (version of the main artifact of C assigned by a VCS)

output: inconsistencies (set of all potential inconsistencies generating by the change)

```
1 begin
2     inconsistencies = Initialize()
3     inconsistencies =
      AnalyzeInconsistencyonAtomicChangeswithSameArtifact(C.artifact,
      C.activity, C.csw, C.project, C.author, C.version, C.inbounds)
4     inconsistencies = inconsistencies ∪
      AnalyzeInconsistencyonAtomicChangeswithDependencyRelatedArtifacts(C.artifact,
      C.activity, C.csw, C.project, C.author, C.outbounds, C.inbounds)
5 end
```

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.

6.6 CSWMS Prototype - Guideline

To use CSWMS, a worker must be a CSWMS user, that is he is supplied with an account that stores 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 the *CSW Manager* to do his tasks (Fig. 6.7-A).

CSW Manager (Fig. 6.7-B) shows the CSWs of the current user (*CSW List*) and the states of the activities in these CSWs (*Todo List* and *Executing Activity 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. From the menu of *CSW Manager*, a user can create a new CSW from scratch (collaborative workflow or model workflow [33]) (Fig. 6.7-C), or as a copy of an existing CSW (collaborative workflow or model workflow), or as an instance of an existing CSW (model workflow).

CSW is edited with *CSW graph editor* (Fig. 6.7-D). Impact Analysis [36] tools can be used to support CSWMS users to identify the impact of a change request to define CSWs more easily and accurately. One does not need to specify all the activities of a CSW at the beginning. He can add new activities to a CSW, or modify an existing activity except the finished activities during the execution of a CSW. For each activity in a CSW, one needs

Function AnalyzeInconsistencyonAtomicChangeswithSameArtifact(C.artifact, C.activity, C.csw, C.project, C.author, C.version, C.inbounds)

```
1  /* Analyze the atomic changes C' having the same main artifact with
      the atomic change C: C.artifact = C'.artifact.                    */
2  begin
3      inconsistencies = Initialize()
4      i = 0
5      /* Query database to get the atomic changes C' having the same
          artifact, same project, but made by different worker with C   */
6      C'Set = SelectC'SameArtifact(C.artifact,C.project,C.author)
7      foreach C' in C'Set do
8          if C'.activity and C.activity are concurrent then
9              if C.revision < C'.revision then
10                 // C.author is changing an outdated version of C.artifact
11                 inconsistencies[i] = createNewPotentialInconsistency(Severe
                   Direct-Conflict, C, C', C.artifact)
12                 i++
13             else if C.revision = C'.revision then
14                 // C.author and C'.author have not yet checked-in C & C'
15                 inconsistencies[i] = createNewPotentialInconsistency(Light
                   Direct-Conflict, C, C', C.artifact)
16                 i++
17             end
18         else if C'.activity happens before C.activity and C'.csw ≠ C.csw then
19             inconsistencies[i] =
               createNewPotentialInconsistency(Direct-Revision-Inconsistency, C,
               C', C.artifact)
20             i++
21             // find WWR Interleaving-Inconsistency in Plan
22             /* Query database to find planned change C"'.  C"'.csw =
                  C'.csw.  C"'.artifact --> C.artifact (C"'.outbounds ⊃
                  C.artifact OR C.inbounds ⊃ C"'.artifact).  C"'.activity
                  has not been executed yet.                            */
23             C"'Set = SelectPlannedChangeC"'WWR(C.artifact, C.project,
               C.inbounds, C'.csw).
24             foreach C"' in C"'Set do
25                 inconsistencies[i] = createNewPotentialInconsistency(WWR
                   Interleaving-Inconsistency in Plan , C', C, C"', C.artifact,
                   C"'.artifact)
26                 i++
27             end
28         end
29     end
30 end
```

Function AnalyzeInconsistencyonAtomicChangeswithDependencyRelatedArtifacts(C.artifact, C.activity, C.csw, C.project, C.author, C.outbounds, C.inbounds)

```
1  /* Analyze the atomic changes C' that have the main artifact
      depending on or depended by the main artifact of the atomic change
      C: C.artifact --> C'.artifact or C'.artifact --> C.artifact     */
2  begin
3      inconsistencies = Initialize()
4      i = 0
5      /* Query database to get the atomic changes C' in the same
          project, but made by different worker with C. C.artifact -->
          C'.artifact (C.outbounds ⊃ C'.artifact) or C'.artifact -->
          C.artifact (C.inbounds ⊃ C'.artifact)                       */
6      C'Set = SelectC'DependencyRelatedArtifacts(C.artifact, C.project,
       C.author, C.outbounds, C.inbounds)
7      foreach C' in C'Set do
8          if C'.activity and C.activity are concurrent then
9              inconsistencies[i] =
               createNewPotentialInconsistency(Indirect-Conflict, C, C',
               C.artifact, C'.artifact)
10             i++
11         else if C'.activity happens before C.activity and C'.csw ≠ C.csw then
12             if C'.artifact −− > C.artifact then
13                 inconsistencies[i] = createNewPotentialInconsistency(Indirect-
                   Revision-Inconsistency, C, C', C.artifact, C'.artifact)

14                 i++
15                 // find RWR Interleaving-Inconsistency in Plan
16                 /* Query database to find planned changes C"'.  C"'.csw =
                      C'.csw.  C"'.artifact --> C.artifact (C"'.outbounds ⊃
                      C.artifact OR C.inbounds ⊃ C"'.artifact).
                      C"'.activity has not been executed yet.          */
17                 C"'Set = SelectPlannedChangeC"'RWR(C.artifact,C.project,
                   C.inbounds, C'.csw).
18                 foreach C"' in C"'Set do
19                     inconsistencies[i] = createNewPotentialInconsistency(RWR
                       Interleaving-Inconsistency in Plan , C', C, C"', C'.artifact,
                       C.artifact, C"'.artifact)
20                     i++
21                 end
22             else if (C.artifact −− > C'.artifact) AND (C'.creationTime >
               C.csw.startTime) then
23                 // Check WWR|RWR Interleaving-Inconsistency
24             end
25         end
26     end
27 end
```

Function AnalyzeInconsistencyonAtomicChangeswithDependencyRelatedArtifacts(C.artifact, C.activity, C.csw, C.project, C.author, C.outbounds, C.inbounds) (cont.)

```
1  begin
2      inconsistencies = Initialize()
3      i = 0
4      /* Query database to get the atomic changes C' in the same
          project, but made by different worker with C. C.artifact -->
          C'.artifact or C'.artifact --> C.artifact                    */
5      C'Set = SelectC'DependencyRelatedArtifacts(C.artifact, C.project,
       C.author, C.outbounds, C.inbounds)
6      foreach C' in C'Set do
7          if C'.activity and C.activity are concurrent then
8              ...
9          else if C'.activity happens before C.activity and C'.csw ≠ C.csw then
10             if C'.artifact --> C.artifact then
11                 ...
12             else if (C.artifact --> C'.artifact) AND (C'.creationTime >
               C.csw.startTime) then
13                 // Check WWR|RWR Interleaving-Inconsistency
14                 /* Query database to find atomic changes C" so that
                      C".csw = C.csw, C".activity happens before
                      C'.activity, and C".artifact --> C'.artifact
                      (C".outbounds ⊃ C'.artifact OR C'.inbounds ⊃
                      C".artifact) or C".artifact = C'.artifact.       */
15                 C"Set = SelectChangeC"(C.csw, C.project, C'.artifact, C'.csw,
                   C'.inbounds)
16                 foreach C" in C"Set do
17                     if C".artifact = C'.artifact then
18                         inconsistencies[i] =
                           createNewPotentialInconsistency(WWR
                           Interleaving-Inconsistency, C", C', C, C".artifact, C.artifact)
19                         i++
20                     else if C.artifact --> C'.artifact then
21                         inconsistencies[i] =
                           createNewPotentialInconsistency(RWR
                           Interleaving-Inconsistency, C", C', C, C".artifact, C'.artifact,
                           C.artifact)
22                         i++
23                     end
24                 end
25             end
26         end
27     end
28 end
```

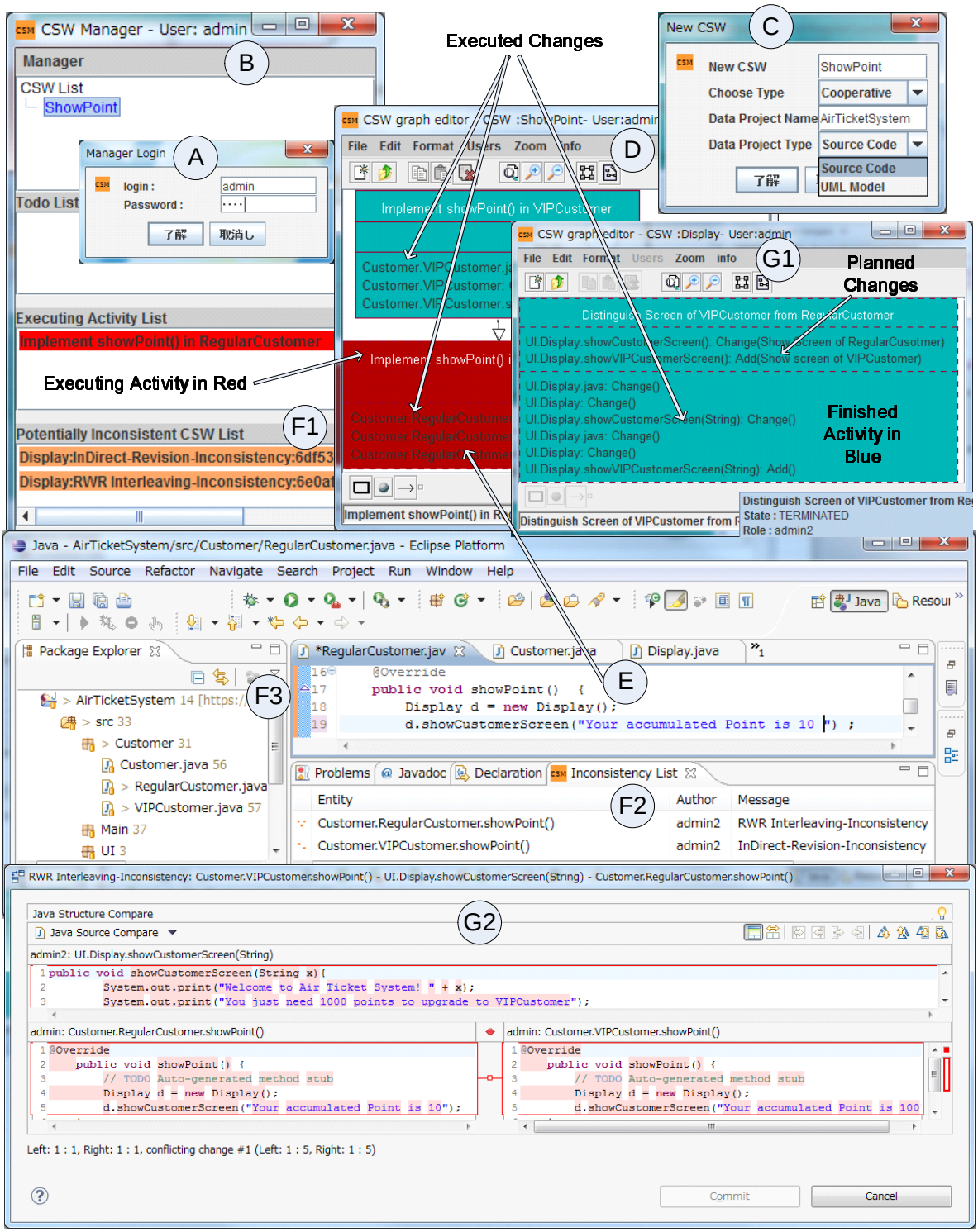


Figure 6.7: CSWMS User Interface. A: Login window for *CSW Manager*. B: *CSW Manager* window. C: *New CSW* Dialog. D: *CSW graph editor* window showing CSW of user *admin*. E. Eclipse IDE with workspace wrapper plugins. F1, F2, and F3: Inconsistency Viewer window on CSW Manager and Eclipse with the warnings of (potential) inconsistencies. G1: *CSW graph editor* window showing CSW of inconsistency-involved user, *admin2*, viewed by *admin*. G2: Inconsistency Viewer windows in Eclipse showing the contents of the changes causing a potential RWR Interleaving-Inconsistency.

to specify the name of the activity. If the user wants, he can also specify the changes he intends to do in an activity, i.e. *planned changes*. The information will be used to predict potential inconsistencies in plan before these changes are done. The changes that really happen to the artifacts in a user's workspace during the execution of an activity, i.e. *executed changes*, will be collected automatically by the **Workspace Wrapper** that is implemented as the plugins of Eclipse (Fig. 6.7-E), and sent to the CSWMS server for inconsistency analysis.

In short, to fulfill a change request, the user first composes a CSW, and then assigns the executors to the activities in the CSW. Each assigned activity will appear in *Todo List* of the worker who is assigned to execute the activity. If a user wants to execute an activity in his *Todo List*, he will select the activity, right click, and choose the *Execute Activity* function. The chosen activity will appear in *Executing Activity List*. When he finishes all the necessary changes, he will select the activity, right click, and choose the *Terminate Activity* function. One must notify the system when he starts and finishes an activity to help the system identify the activity that a change belongs to.

When an inconsistency notification appears in *Potentially Inconsistent CSW List* of the *CSW Manager* window (Fig. 6.7-F1) or Eclipse IDE (Fig. 6.7-F2), one should open it to understand the situation. He can ignore by choosing the *Ignore* function, or choose the *Detail* function to learn more about the inconsistency. Currently, we do not give a solution to the detected inconsistency, but support the users in solving the inconsistency by supplying them as much as possible the information about the context of the inconsistency, including the contents of the changes causing the inconsistency (Fig. 6.7-G2) and the CSWs of these changes (Fig. 6.7-D,G1). After finishing inconsistency resolution, the user should update the state of the inconsistency to *Solved*. Updating the progress of inconsistency resolution helps reduce false warnings from the system.

6.7 How CSWMS Supports Workers in Inconsistency Awareness

The previous works support the workers in detecting emerging inconsistencies caused by the ongoing changes in the workspaces, namely the conflicts, only. Regarding the changes committed to the repository, a worker must recognize the new changes of other workers and their impact to his work, by regularly synchronizing his workspace with the remote repository, or reading tons of emails automatically sent by the VCSs for new check-in notifications, and then investigating all the new changes. On the other hand, the solution in which the worker must guarantee the consistency of the effects by his changes by notifying his changes to an artifact to the current and potential users of the artifact by himself to avoid inconsistencies is also tricky.

CSWMS aims to reduce the load of the workers and to handle inconsistencies more effectively. We notify the workers of the contexts of the changes causing the (potential) inconsistencies rather than just the changes themselves. The contexts of the changes can help them understand the problem more easily to have more proper and timely decisions. Instead of finding the related workers, specifying the changes, and notifying them of his changes by himself, a worker only needs to specify his change process, i.e. the CSW, using *CSW graph editor* supported by CSWMS. Managing the execution of CSWs, collecting and analyzing information to detect (potential) inconsistency, and notifying the workers

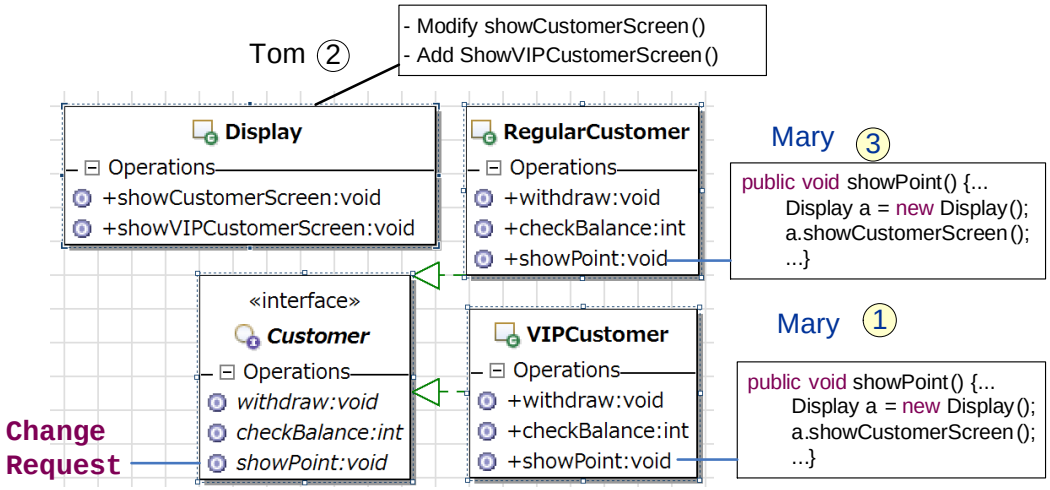


Figure 6.8: CSWMS effectiveness - Illustrating example for Indirect-Revision-Inconsistency and RWR Interleaving-Inconsistency

of (potential) inconsistencies along with their contexts, including the changes causing the inconsistencies and the CSWs of these changes, are automatically handled by our system. Hence, CSWMS can help the workers detect and resolve many types of inconsistency rather than conflicts only, more quickly and efficiently.

To illustrate the effectiveness of CSWMS that combines the workspace awareness and context awareness techniques, as opposed to other approaches using the workspace awareness technique only [21, 24, 20, 25, 26, 28], we will show some examples of getting benefits from CSWMS that were not provided by the previous studies. We simulate the situations, which were described in the previous chapters as the illustrating examples for the patterns of inconsistency (Section 3.2) and the motivating example for our approach (Section 2.2), to contrast a traditional system with CSWMS.

6.7.1 Potential Indirect-Revision-Inconsistency and RWR Interleaving-Inconsistency

Fig. 6.8 shows the situation that was described as the motivating example in Section 2.2.

In the morning, Mary, whose username in CSWMS is *admin*, uses *CSW Manager* and *CSW graph editor* to define a CSW *ShowPoint* for implementing the change request adding the *showPoint()* method to the *Customer* interface. The *ShowPoint* CSW has two activities, *Implement showPoint() in VIPCustomer* and *Implement showPoint() in RegularCustomer*. She executes the activity *Implement showPoint() in VIPCustomer* first. Although she does not specify any planned changes, thanks to the *Workspace Wrapper* plugin in Eclipse, the atomic changes in Mary's workspace during the execution of the first activity are still sent to the server automatically (*executed changes* of the first activity in Fig. 6.9-E2). When the first activity finishes, she delays the execution of the second activity until that night, because she has an urgent meeting with a customer.

Indirect-Revision-Inconsistency warning for Tom: In the afternoon, Tom, whose username is *admin2*, starts modifying the *Display* class by defining a *Display* CSW with one activity. When he specifies the planned changes, he receives a warning of potential Indirect-Revision-Inconsistency about the change context of Mary, the *ShowPoint* CSW.

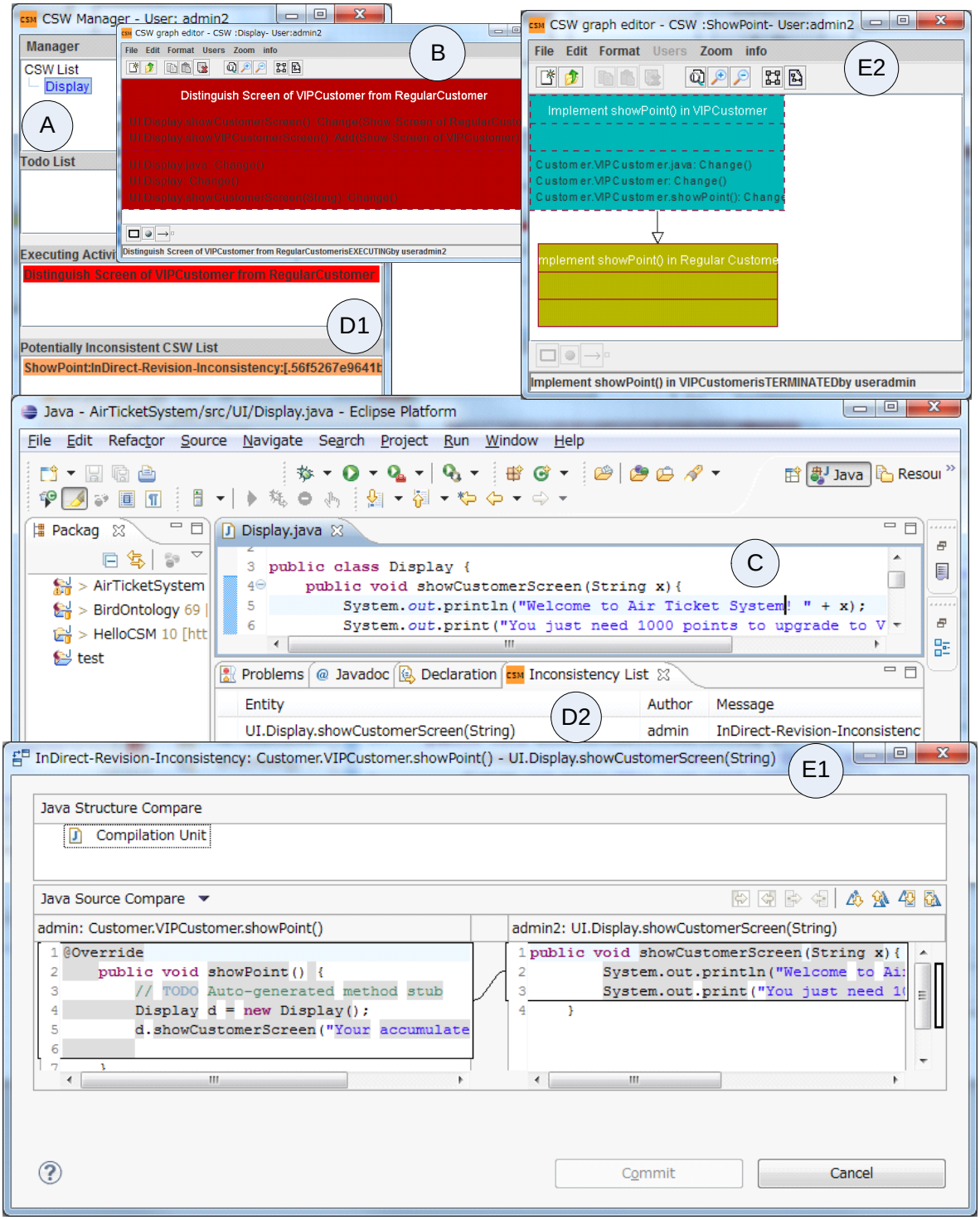


Figure 6.9: Example of a potential Indirect-Revision-Inconsistency detected by CSWMS. A: *CSW Manager* window of *admin*2. B: The *Display* CSW of *admin2*. C: *admin*2 is modifying the *Display* class using Eclipse IDE. D1 & D2: Warning for Indirect-Revision-Inconsistency appears in *CSW Manager* and Eclipse IDE of *admin*2. E1: Contents of the changes causing the potential Indirect-Revision-Inconsistency are shown in Eclipse IDE of *admin*2. E2: The *ShowPoint* CSW of *admin* is viewed by *admin2* to understand the context of the inconsistency.

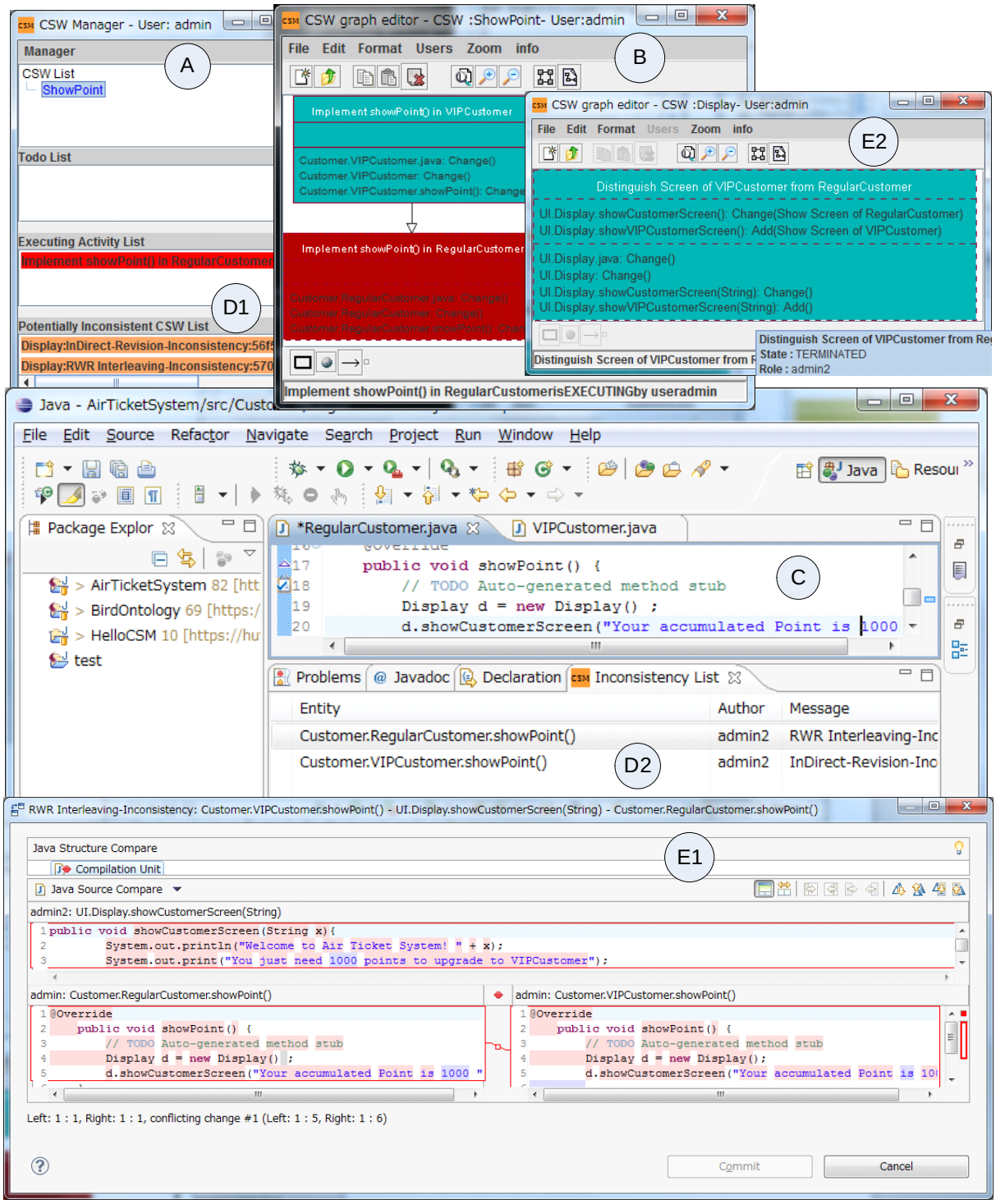


Figure 6.10: Example of a potential RWR Interleaving-Inconsistency detected by CSWMS. A: *CSW Manager* window of *admin*. B: The *ShowPoint* CSW of *admin*. C: *admin* is implementing the *showPoint*() method for the *RegularCustomer* class using Eclipse IDE. D1 & D2 : Warning for WWR Interleaving-Inconsistency appears in *CSW Manager* and Eclipse IDE of *admin*. E1: Contents of the changes causing the potential RWR Interleaving-Inconsistency are shown in Eclipse IDE of *admin*. E2: The *Display* CSW of *admin2* is viewed by *admin* to understand the context of the inconsistency.

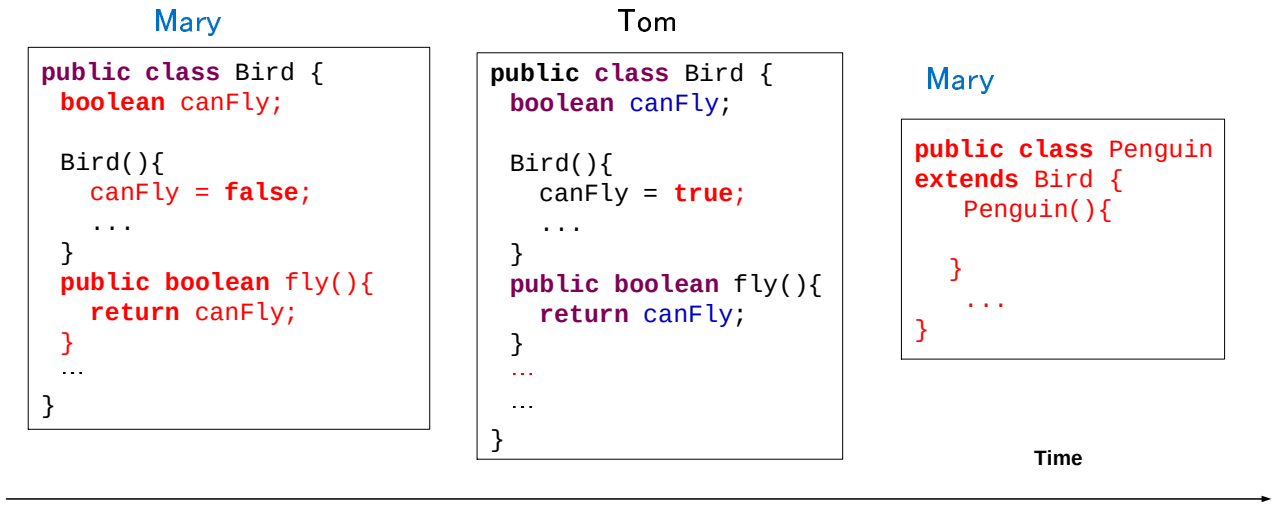


Figure 6.11: CSWMS effectiveness - Illustrating example for the Direct-Revision-Inconsistency and WWR Interleaving-Inconsistency patterns

Assuming that he ignores the warning and continues executing the activity by modifying the *Display* class. A warning of Indirect-Revision-Inconsistency replacing the previous warning is shown. If he opens the warning, CSWMS will show him the context of the potential inconsistency which includes the *Display* CSW of Tom, the *ShowPoint* CSW of Mary, and the source codes of the changes causing the potential inconsistency (Fig. 6.9). By examining it, Tom can recognize that his change affects the finished work of Mary, implementing the *showPoint*() method in the $VIPCustomer$ class, and may affect her future work, *Implement showPoint() for RegularCustomer*. Assuming that Tom again ignores it.

Indirect-Revision-Inconsistency and RWR Interleaving-Inconsistency warnings for Mary: At night, when Mary logs-in to CSWMS, there is a warning about Indirect-Revision-Inconsistency at the bottom of her *CSW Manager*. Assuming that she ignores the warning and starts the second activity, *Implement showPoint() for RegularCustomer*. When she begins coding the *showPoint*() method of the *RegularCustomer* class using Eclipse IDE, another warning of RWR Interleaving-Inconsistency is shown in the *Potentially Inconsistent Project List* window and Eclipse (6.10-D1,D2), and the CSW of Tom, *Display*, which is the cause of this inconsistency, is opened (6.10-E2). When she double-clicks on the latest warning in her inconsistency list in Eclipse, the window that compares the source codes of the atomic changes causing the inconsistency is shown (Fig. 6.10-E1). With the supplied information of the inconsistency context, Mary could recognize the modification of the *showCustomerScreen*() method and the appearance of the $showVIPCustomerScreen()$ method. By understanding the change context of each other, she could coordinate with Tom to solve the inconsistency quickly and easily.

6.7.2 Potential Direct-Revision-Inconsistency and WWR Interleaving-Inconsistency

Fig. 6.11 shows the situation that is similar to the illustrating example for the WWR Interleaving-Inconsistency pattern in Section 3.2.8.

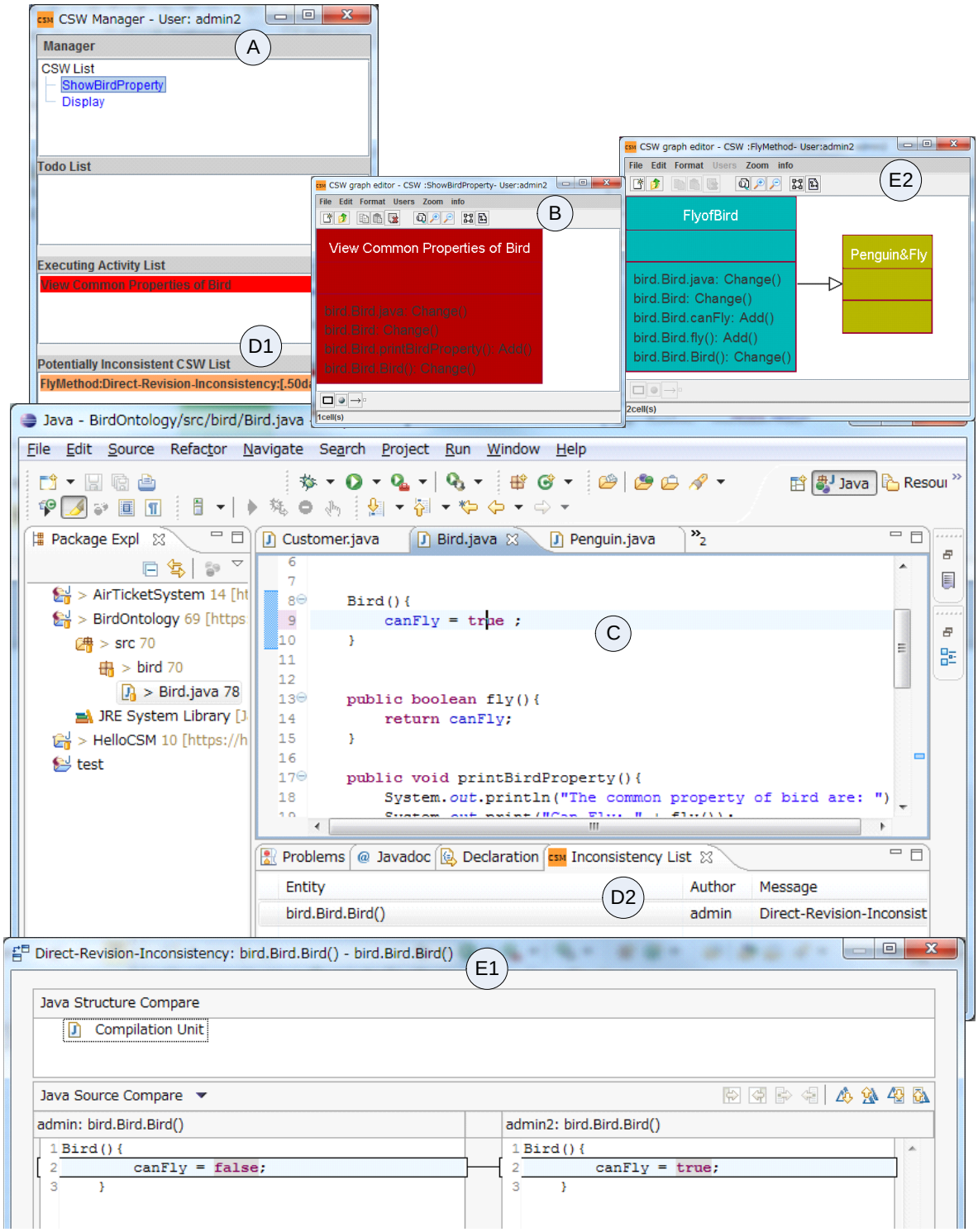


Figure 6.12: Example of a potential Direct-Revision-Inconsistency detected by CSWMS. A: *CSW Manager* window of *admin*2. B: The *ShowBirdProperty* CSW of *admin2*. C: *admin*2 is modifying the *Bird* class using Eclipse IDE. D1 & D2: Warning for Direct-Revision-Inconsistency appears in *CSW Manager* and Eclipse IDE of *admin*2. E1: Contents of the changes causing the potential Direct-Revision-Inconsistency are shown in Eclipse IDE of *admin*2. E2: The *FlyMethod* CSW of *admin* is viewed by *admin2* to understand the context of the inconsistency.

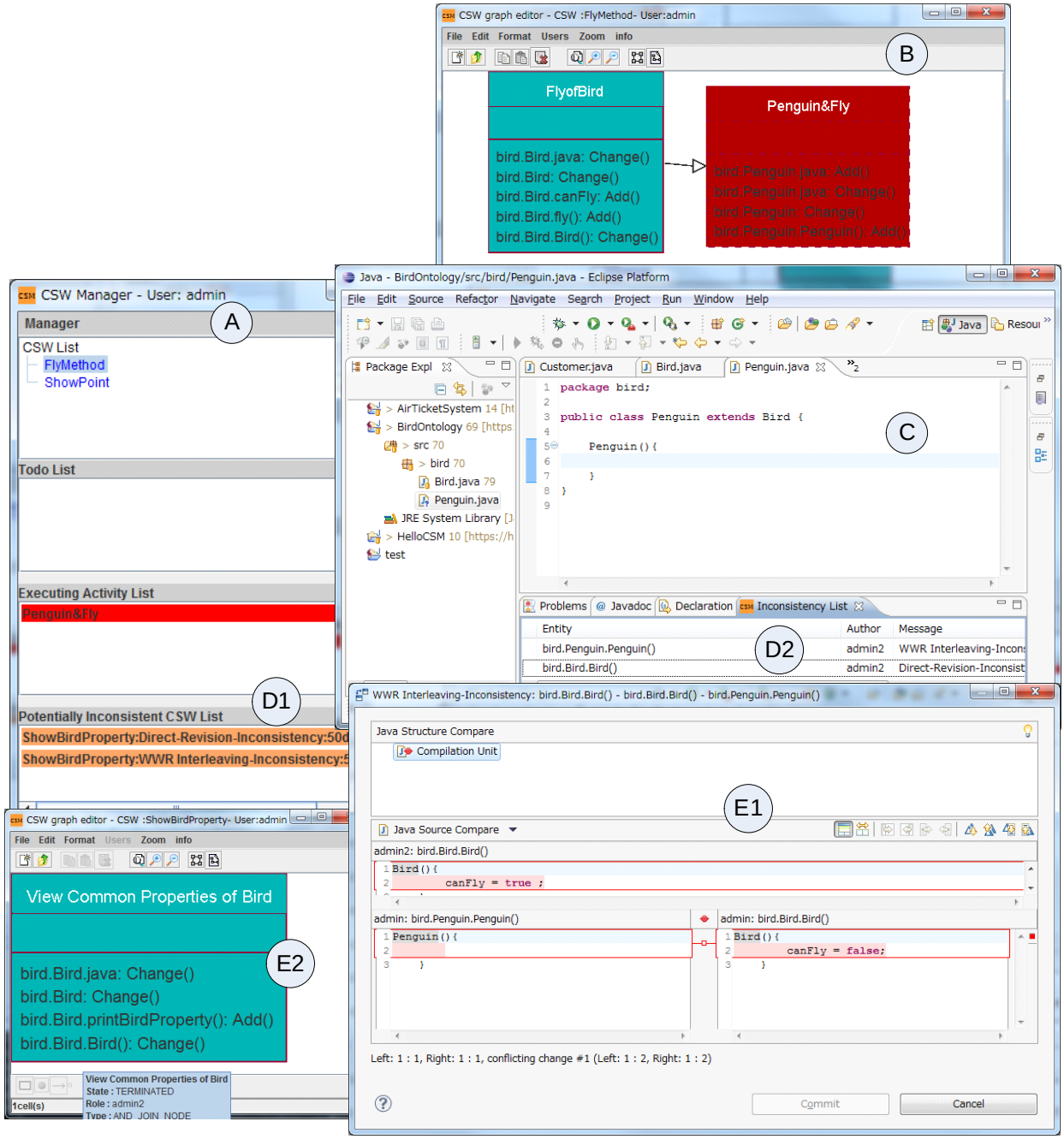


Figure 6.13: Example of a potential WWR Interleaving-Inconsistency detected by CSWMS. A: *CSW Manager* window of *admin*. B: The *FlyMethod* CSW of *admin*. C: *admin* is coding the *Penguin* class using Eclipse IDE. D1 & D2: Warning for WWR Interleaving-Inconsistency appears in *CSW Manager* and Eclipse IDE of *admin*. E1: Contents of the changes causing the potential WWR Interleaving-Inconsistency are shown in Eclipse IDE of *admin*. E2: The *ShowBirdProperty* CSW of *admin2* is viewed by *admin* to understand the context of the inconsistency.

First, Mary, whose username is *admin*, defines the *FlyMethod* CSW for implementing the change request that represents the ability to fly of birds. Because she does not have a clear plan for what she will do, she defines *FlyMethod* with one activity only, the *FlyofBird* activity, and executes it. She adds the field *canFly* to denote the ability to fly of birds. In the constructor of the *Bird* class, she sets the default value of *canFly* to $false$, because she thinks that this helps specify the bird species more conveniently. With flying birds, for example Eagle, one needs to describe more information about this ability such as speed. Therefore, one can change the value of *canFly* to *true* in the constructor, and override the $fly()$ method to add more information. With flightless birds, for example Penguin, one can ignore this function. Because she has an appointment with a customer for lunch, she finishes the current activity. To not forget the ongoing work after coming back from lunch, Mary adds to the *FlyMethod* CSW a new activity, the *Penguin&Fly* activity, as a successor of the *FlyofBird* activity, before leaving the office.

Direct-Revision-Inconsistency warning for Tom: While Mary is having lunch with the customer, Tom, whose username is *admin2*, implements the change request for displaying the common properties of birds. He defines the *ShowBirdProperty* CSW with one activity, the *View Common Properties of Bird* activity, first and starts it. When he adds the new methods to the *Bird* class, for example the *printBirdProperty()* method, he recognizes that the *fly()* method returns the *false* value, because the *canFly* field is set to *false* in the constructor of *Bird*. In his opinion, most birds can fly. Therefore, he changes the value of *canFly* to *true*. As a result, a warning of Direct-Revision-Inconsistency is shown. If he opens the warning, CSWMS will show him the context of the potential inconsistency which includes the *ShowBirdProperty* CSW of his, the *FlyMethod* CSW of Mary, and the source codes of the changes causing the potential inconsistency (Fig. 6.12). By examining it, Tom can recognize that his changes may affect Mary's work. He can delay his changes after discussing with Mary, or keep his opinion but add more explanations about his work to his CSW to help Mary understand his intention. Assuming that Tom ignores the warning. He finishes his activity before Mary returns to the office.

Direct-Revision-Inconsistency and WWR Interleaving-Inconsistency warnings for Mary: Coming back from lunch, Mary continues her work. There is a warning about Indirect-Revision-Inconsistency at the bottom of her *CSW Manager*. If she investigates the warning, she can realize that the change of Tom affects her work, and she can adjust her implementation plans to make the system work correctly. Assuming that she continues her work without paying attention to the warning. Persisting with her original intention, she starts the *Penguin&Fly* activity and begins coding the *Penguin* class. When she is editing its constructor, a warning of WWR Interleaving-Inconsistency appears. If she does not ignore it again, she could understand what is happening by examining the context of the potential inconsistency (Fig. 6.13). She can solve the situation by setting the value of the field *canFly* to *false* in the constructor of *Penguin*, or negotiating with Tom to change the value of *canFly* in the constructor of *Bird* to *false* as her original intention.

6.8 Summary

We have presented the development and implementation of CSWMS that realizes our approach to solve the inconsistency problem in collaborative software development.

CSWMS has the advantage of warning workers as early as possible (potential) inconsistencies to help the workers prevent or solve them before their effects go further. CSWMS can detect exactly Intra|Inter-Direct-Conflicts and the potential cases of the remaining patterns of the patterns of inconsistency. However, in the case of the semantic inconsistencies, CSWMS cannot ensure the detected potential inconsistencies will lead to real inconsistencies or not. Since the semantics of the changes are difficult to express in a formal way, the detection of such problems is challenging and requires the intervention of the workers involved. In the case of a false warning, it interrupts the workers from their work and takes time to examine the falsely reported inconsistency. However, by applying the fine-grained analysis, we can reduce the false warnings. Moreover, by supplying the workers with the CSWs of the changes that cause the reported inconsistency, we can help them understand the context of the inconsistency and skip the false alarm quickly and easily. In addition, even if an inconsistency does not happen, the warnings of potential inconsistencies still contribute to increasing the awareness of a worker about the works of his co-workers. This directs the worker to make correct changes and avoid unexpected impacts on the changes of other workers.

Chapter 7

Performance Evaluation and Discussion

In this chapter, we first make a comparison between our work and the related studies toward the inconsistency problem in collaborative software development. We then evaluate performance of the inconsistency detection algorithm to examine its scalability. Some discussions about the effectiveness of our research and the challenges in evaluating its effectiveness are also given.

7.1 Comparison with Related Studies

The novelty of our approach lies in addressing the inconsistency at the view of the change processes containing the changes rather than the individual changes. Our inconsistency awareness is an extension of the related studies in this field that used the workspace awareness technique only. By combining the workspace awareness technique with the context awareness technique, we can recognize many types of inconsistencies, in addition to conflicts. We can also supply much information about the context of the changes causing a (potential) inconsistency to the workers, rather than the changes causing (potential) conflicts only like the previous studies. Table 7.1 summarizes the main features of the related works, and shows the relationships between them and our study.

7.2 Performance Evaluation of Inconsistency Detection Algorithm

In this section, we conduct performance evaluation to examine the scalability of our inconsistency detection algorithm.

In the current prototype, to reduce unnecessary atomic changes, the *executed changes*, i.e. the atomic changes collected by monitoring the workspaces of workers, are used only when the workers have not yet checked-in their changes. If the workers have checked-in their changes, we transform the textual changes in these revisions into the structured changes, and use these atomic changes, named *committed changes*, instead of the executed changes happened in the workspaces of the workers in the time intervals before these commits were made. In other words, the executed changes reflect the interim states of the changed artifacts, and the committed changes show the final states of the

Table 7.1: Summary of related works in inconsistency awareness

References	Approach	Detected Inconsistency	Change Granularity	Supplied Information about Inconsistency
Crystal [26]	Merge local repositories of distributed VCSs	Pending conflicts	Coarse	Type of conflicts
WeCode [27]	Continuously merge Eclipse workspaces	Emerging conflicts	Fine	Changes causing conflicts
TUKAN [19]	Build a spatial model for the shared source code under development	Emerging conflicts	Fine	Changes causing conflicts
CollabVS [20]	Monitor Visual Studio workspaces	Emerging conflicts	Fine	Changes causing conflicts
Palantir [21]	Monitor Eclipse workspaces	Emerging conflicts	Medium	Changes causing conflicts
Lighthouse [23]	Show emerging designs that update changes on the workspaces of workers	-	Fine	Which source code entities are modified and by whom
CASI [24]	Show emerging designs that update changes on the workspaces of workers + Sphere of influence	-	Fine	Which source code entities are modified and by whom + Which source code entities are influenced by their changes
Syde [25]	Monitor Eclipse workspaces	Emerging conflicts	Fine	Changes causing conflicts
CSWMS	*Monitor Eclipse workspaces + Manage execution of change processes + Transform textual revisions in VCS into structured changes*	*Emerging conflicts + Direct/Indirect-Revision-Inconsistency + RWR/WWR Interleaving-Inconsistency*	*Fine*	*Changes causing inconsistencies + Change processes of these changes*

Table 7.2: Number of queries to find an inconsistency

Inconsistency Type	**Number of Queries**
Intra|Inter-Direct-Conflict	1
Intra|Inter-Direct-Conflict	1
Direct-Revision-Inconsistency	1
Indirect-Revision-Inconsistency	1
RWR-Interleaving-Inconsistency	1 + t
WWR-Interleaving-Inconsistency	1 + t
Total number of queries: 1 + 1 + 1 + 1 + t = 4 + t - Indirect-Revision-Inconsistency, RWR Interleaving-Inconsistency, and WWR Interleaving-Inconsistency share the query for finding the atomic changes C' whose main artifacts depend on or are depended by the main artifact of C. - t is the number of the atomic changes C' with the creation time later than the start time of Cs CSW and the main artifacts depended by the main artifact of C. - RWR Interleaving-Inconsistency and WWR Interleaving-Inconsistency share t queries to find the atomic changes C" whose main artifacts are depended by or the same with the main artifact of one among the t atomic changes C'.	

artifacts. Currently, we store the atomic changes of each Java project in two tables: **WorkspaceChanges** and **SVNChanges**. **WorkspaceChanges** contains the executed changes, and **SVNChanges** contains the committed changes. We also use another table, **OngoingChanges**, for storing the executed changes that have not been committed yet, named *ongoing changes*. Upon receiving an executed change C, we search the **OngoingChanges** table to detect Intra|Inter-Direct-Conflict and Intra|Inter-Indirect-Conflict. The changes stored in **OngoingChanges** are also stored in **WorkspaceChanges**, however storing only ongoing changes helps reduce the search space. **SVNChanges** is used for detecting Direct-Revision-Inconsistency, Indirect-Revision-Inconsistency, and RWR|WWR Interleaving-Inconsistency.

Since querying the database accounts for most of the execution time of the inconsistency detection algorithm, the performance evaluation is focused on the execution time of each type of query for searching the patterns of inconsistency in the **SVNChanges** and **OngoingChanges** tables. We assume that there is a team of eight workers working on a Java project. We generate the data for the experiments based on the source code of the SVNKit[1] project. SVNkit is chosen because it is a well-known open source Java API library for Subversion [9], a leading and fast growing open source version control system. We convert 87 revisions of *svnkit*[2] committed by four developers from June 2012 to June

[1]http://svnkit.com/index.html
[2]http://svn.svnkit.com/repos/svnkit/trunk/svnkit/

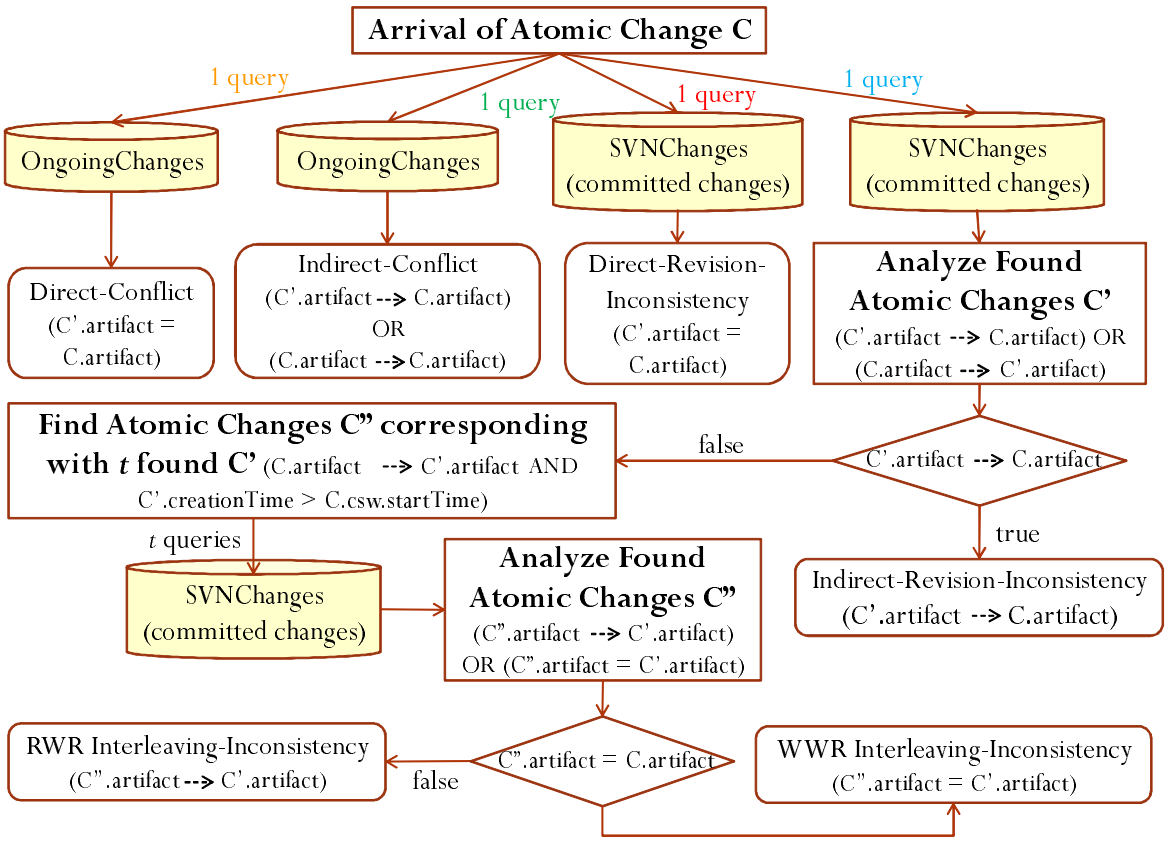


Figure 7.1: Inconsistency detection

2013, and 66 revisions of *svnkit-test*[3] committed by four developers from June 2012 to June 2013, to atomic changes. In so doing, we have 1043 and 751 atomic changes from *svnkit* and *svnkit-test*, respectively. Then, we add essential information, such as CSWs and change activities, and scale up the generated data to fit the specific requirements of each experiment. The experiments are conducted with an Intel Core i7-2600K CPU 3.4GHz machine with 8 GB memory, and MySql Workbench 5.2 CE.

Table 7.2 shows the number of queries on the database to find the patterns of inconsistency, and Fig. 7.1 explains how the queries are used to detect the patterns of inconsistency. Because we use the **OngoingChanges** table for detecting conflicts and the **SVNChanges** table for detecting the remaining patterns, we need one query on **OngoingChanges** for Direct-Conflict, one query on **OngoingChanges** for Indirect-Conflict, and one query on **SVNChanges** for Direct-Revision-Inconsistency. To reduce the cost, Indirect-Revision-Inconsistency, RWR-Interleaving-Inconsistency, and WWR-Interleaving-Inconsistency share the query for finding the atomic changes whose main artifacts depend on or are depended by the main artifact of C. Among the atomic changes returned by this query, let t be the number of the atomic changes with the creation time later than the start time of C's CSW and the main artifacts depended by the main artifact of C. Then, RWR Interleaving-Inconsistency and WWR Interleaving-Inconsistency also share t queries to find C" corresponding with these t atomic changes. Therefore, the cost for finding RWR|WWR Interleaving-Inconsistency depends on the value of t.

[3]http://svn.svnkit.com/repos/svnkit/trunk/svnkit-test/

Table 7.3: Execution time of queries on **OngoingChanges** for detecting conflicts

Number of Ongoing Changes	**Time for One Query (Millisecond)**	
	Find Direct-Conflict	**Find Indirect-Conflict**
2086	22 (Not found: 10)	43 (Not found: 18)
4172	32 (Not found: 15)	68 (Not found: 31)
8344	52 (Not found: 22)	119 (Not found: 55)
16688	101 (Not found: 41)	245 (Not found: 108)
33376	185 (Not found: 76)	465 (Not found: 206)

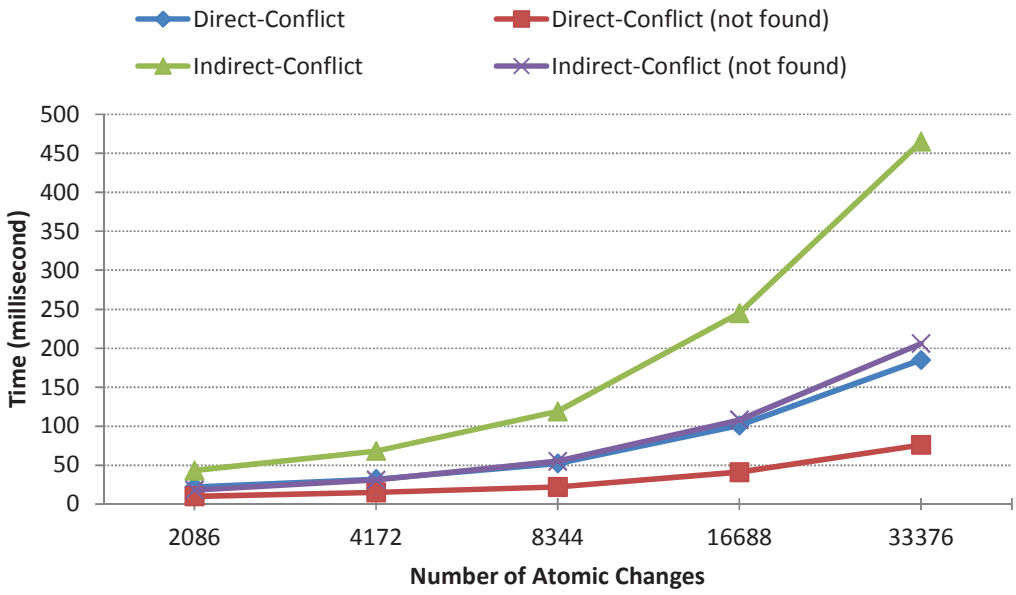


Figure 7.2: Execution time of queries on **OngoingChanges** for detecting conflicts

Regarding Intra|Inter-Direct-Conflict and Intra|Inter-Direct-Conflict, the search space is the **OngoingChanges** table, with the atomic changes that are made by the executing activities and have not yet committed. Because the number of activities that are executing at the same time by the team members is usually small, our experiments are set up with 16 executing activities. Table 7.3 and Fig. 7.2 show the execution time of each type of query that is proportional to the number of ongoing changes made by these activities.

With the remaining patterns, Direct-Revision-Inconsistency, Indirect-Revision-Inconsistency, and RWR|WWR Interleaving-Inconsistency, the search space is the **SVNChanges** table containing the committed changes. The execution time of the queries depends on the number of CSWs that have been finished or are being executed within a predefined time

Table 7.4: Execution time of queries on **SVNChanges** for detecting Direct-Revision-Inconsistency, Indirect-Revision-Inconsistency, and RWR|WWR Interleaving-Inconsistency

Number of CSWs	**Time for One Query (Millisecond)**		
	Find Direct-Revision-Inconsistency	**Find C' of Indirect-Revision-Inconsistency and RWR|WWR Interleaving-Inconsistency**	**Find C" of RWR|WWR-Interleaving-Inconsistency**
8 (1043 atomic changes, 87 revisions)	19 (Not found: 8)	32 (Not found: 15)	13 (Not found: 9)
16 (2086 atomic changes, 174 revisions)	22 (Not found: 10)	43 (Not found: 18)	18 (Not found: 13)
20 (2837 atomic changes, 240 revisions)	25 (Not found: 11)	55 (Not found: 22)	21 (Not found: 16)
40 (5674 atomic changes, 480 revisions)	42 (Not found: 18)	105 (Not found: 38)	35 (Not found: 24)
60 (8511 atomic changes, 720 revisions)	56 (Not found: 25)	146 (Not found: 54)	46 (Not found: 34)
80 (11348 atomic changes, 960 revisions)	67 (Not found: 28)	182 (Not found: 64)	59 (Not found: 41)
100 (14185 atomic changes, 1200 revisions)	80 (Not found: 34)	220 (Not found: 80)	71 (Not found: 50)
200 (28370 atomic changes, 2400 revisions)	148 (Not found: 58)	424 (Not found: 151)	131 (Not found: 91)
300 (42555 atomic changes, 3600 revisions)	218 (Not found: 85)	629 (Not found: 225)	190 (Not found: 135)

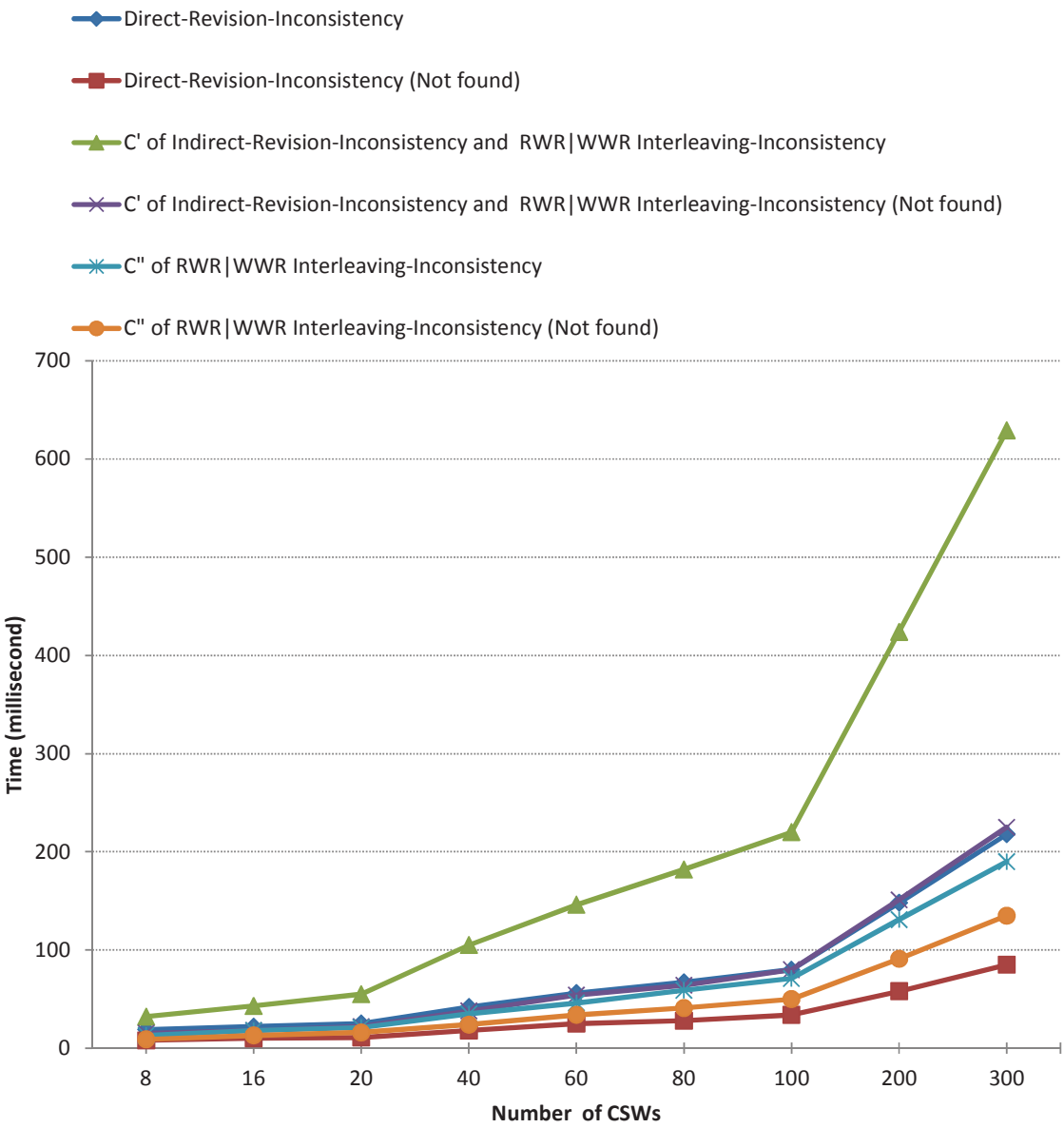


Figure 7.3: Execution time of queries on **SVNChanges** for detecting Direct-Revision-Inconsistency, Indirect-Revision-Inconsistency, and RWR|WWR Interleaving-Inconsistency

Table 7.5: Size of data in real projects [56]

Project	Number of Files	Number of Revisions	Number of atomic changes	Examined Time Interval
Compare	154	2953	17263	May 2001 – Sep 2010
jFace	378	5809	22203	Sep 2002 – Sep 2010
JDT DB Jdi	144	1936	6121	May 2001 – July 2010
JDT DB Eval	105	1610	6091	May 2001 – July 2010
JDT DB Model	98	2546	12566	May 2001 – July 2010
Resource	274	6558	28946	May 2001 – Sep 2010
Team Core	169	1995	4607	Nov 2001 – Aug 2010
CVS Core	188	5448	23301	Nov 2001 – Aug 2010
DB Core	187	3033	12342	May 2001 – Sep 2010
jFaceText	312	4980	23633	Sep 2002 – Oct 2010
Update Core	275	6379	27465	Oct 2001 – Jun 2010
DB UI	788	10909	57075	May 2001 – Oct 2010
JDT DB UI	381	5395	28956	Nov 2001 – Sep 2010
Help	110	999	5919	May 2001 – May 2010
JDT Compiler	322	19466	171965	Jun 2001 – Sep 2010
JDT Dom	157	6608	32699	Jun 2001 – Sep 2010
JDT Model	420	16892	90128	Jun 2001 – Sep 2010
JDT Search	115	5475	44372	Jun 2001 – Sep 2010
OGSI	395	6455	38203	Nov 2003 – Oct 2010

interval. We assume that the average number of the activities of each CSW is about 12, corresponding with 12 revisions. Table 7.4 and Fig. 7.2 show the execution time of each type of query that is proportional to the number of CSWs.

Based on the results of the experiments, with 33376 ongoing changes in the **OngoingChanges** table and 42555 committed changes of 300 CSWs in the **SVNChanges** table, the time for detecting the patterns of inconsistency is *185 + 465 + 218 + 629 + 190*t = 1497 + 190*t* milliseconds. Because $t \leq$ *(number of CSWs concurrent with the CSW of C) * length(C.outbounds)*, in the worst case, the execution time of the algorithm is *1497 + 190*(number of CSWs concurrent with the CSW of C)*length(C.outbounds)* $\leq$ *1497 + 190*(number of CSWs)*length(C.outbounds) = 1497 + 190*300*length(C.outbounds) = 1497 + 57000*length(C.outbounds)*. As low coupling is required in software development, we expect that the number of outbound artifacts of an artifact is not too large, for example under 10, and rarely over 100. If length(C.outbounds) is 100, the execution time is 5701497 milliseconds, which is about 95 minutes. In this case, more powerful machine and DBMS are necessary. Fortunately, this situation is thought to rarely happen in practice, because both the number of concurrent CSWs in a Java project and the number of

outbound artifacts of an atomic change are not large. For example, in the case of our experimental setting, a team of eight workers will have less than 20 concurrent CSWs. With the number of outbound artifacts of C under 10, we expect that t is smaller than 100 in most cases. In this case, the execution time is under 20497 millisecond, or about 20.5 seconds, which is acceptable. We can also set an upper bound of t to ensure that the execution time of the algorithm is always reasonable.

In order to have a comparative view on the data size, we use the data published in [56] to compare the size of data used in our experiments with that of the real projects. Table 4 describes the sizes of 19 plugin projects of the Eclipse platform. In ten years, 16/19 projects have the number of revisions below 7000, as against 3600, the highest number of revisions used in our experiments, and 15/19 projects have the number of atomic changes below 40000, compared with 42555, the highest number of atomic changes used in our experiments. Because we limit the search space to CSWs executed within a predefined time interval, for example 3 months, we expect that our experiments can reflect the real state of typical projects in practice, and our inconsistency detection algorithm can be applied to practical situations.

7.3 Discussion

7.3.1 Effectiveness of the proposed approach

We have developed the model and the tool to realize a Change Support Environment able to detect in advance the (potential) inconsistencies that could not be detected by the previous works, and provide the contexts of the changes causing the inconsistencies rather than the changes only like the previous work. Therefore, our work can help the workers in preventing, detecting, and resolving inconsistencies in collaborative software development more effectively.

Due to time and resource constraints, the current prototype of CSWMS is still a simple laboratory prototype. However, basically, it can represent the effectiveness of our work.

- **Accuracy of inconsistency detection:** Regarding the accuracy of the warnings of inconsistency, i.e. whether the reported situation will lead to a real inconsistency or not,
  - If the warnings of inconsistency are correct, namely the reported situation leads to a real inconsistency, it is clear that our work is effective.
  - If the warnings of inconsistency are incorrect, namely the reported situation does not lead to a real inconsistency, it interrupts the workers from their work and takes time to examine the falsely reported inconsistency. This problem is also faced by the previous studies [20, 21, 23, 24, 25]. 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 CSWs of the changes that cause the reported inconsistency, to help them understand the context of the inconsistency and skip the false alarm quickly and 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.

- **Permanent benefits:** Regardless of the accuracy of the warnings of inconsistency, through the warnings of potential inconsistencies and showing the CSWs of the workers involved, our study can contribute to increasing the awareness of a worker about the works of his co-workers. This awareness helps 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.

To sum up, our research, which notifies the worker in advance of (potential) inconsistencies along with the context of an inconsistency, including the contents of the changes causing the inconsistency and their CSWs, 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 the related workers makes the collaborations among the workers easier. Therefore, the advantages of our approach outweigh its disadvantages.

7.3.2 Scalability of the proposed approach

We have tested the current prototype of CSWMS with the small projects simulating the situations given in the motivating example and demonstrations of the patterns of inconsistency. In practice, the number of software artifacts and workers is much larger, and the complexity of software artifacts and their dependencies also increases. Despite the differences in the scale of the tested projects and real software systems, our approach and the CSWMS are still applicable to practice, because the problems we must solve are basically the same.

- The first problem is to monitor the workspaces of the workers to find out the ongoing changes. Because most workers develop software on integrated development environments (IDEs), and these IDEs support tracking the interactions of users with the environments themselves, the monitoring problem is feasible and not affected much by the scale of the monitored software. For example, the current prototype develops the workspace monitoring component as plugins of Eclipse development environment. By registering the resource change events describing the specifics of the changes (or set of changes) that have occurred in the workspace, and using libraries and tools supplied by Eclipse, for example Eclipse JDT and Eclipse AST, we can detect ongoing changes with a fine granularity. As this component is developed based on the functions provided by Eclipse, there should be no concern about the scale of the developed software.

- The second problem is about analyzing atomic changes and the CSWs to detect inconsistencies. The inconsistency detection algorithm will be affected by the size of the data it examines and the number of patterns of inconsistency. In the current prototype, we use the search function of the database management system (DBMS), MySQL, to find the matching patterns. Therefore, updating the database system to a more powerful DBMS, for example Oracle, can speed up the searching process. We also limit the search space to the CSWs and atomic changes defined or executed within a predefined time interval. Moreover, by transforming SVN textual revisions into *committed changes*, and using these atomic changes in addition to the *ongoing*

changes, we can reduce the search space to an acceptable number. As we have presented in Section 7.2, our inconsistency detection algorithm is feasible to most practical situations. In the worse cases, more powerful machine and DBMS can solve the performance problem.

In summary, with more investments, we can develop a mature version of CSWMS that satisfies the requirements of real software developments.

7.3.3 How to evaluate exactly the effectiveness of the patterns of inconsistency and the proposed approach?

Using the current prototype of CSWMS, we can detect the patterns of inconsistency, which the previous works could not detect, from the situations described in this dissertation, such as the motivating example and examples of the patterns of inconsistency, and other similar situations. However, evaluating exactly the effectiveness of our work by traditional evaluation methods, such as measuring the precision and accuracy of the detected inconsistency, is very difficult because:

- Lacking a standard benchmark.

  We do not have data of a collaborative software development in industry. In the case of open source software, they do not supply information about their development processes that are very important to identify the existences of real inconsistencies. Except for direct-conflict, which can be recognized by examining the history merges [37], [26], information about other types of inconsistency does not appear in the log file of VCSs. Therefore, extracting information about the other types of inconsistency from the VCSs is still an open research question.

- It is difficult to represent the semantics of a change in a formal way.

  The sufficient condition of an inconsistency varies from case to case, and depends on the consistency among the semantics of the changes and the intentions of the worker making the changes. This consistency can only be decided by the workers who make the changes. As a result, it is very difficult to apply conventional accuracy measurement techniques in this study. The related studies in inconsistency awareness such as [25, 27, 21] did not measure the precision and accuracy of their conflict detection method either. They evaluated the effectiveness of their methods based on the feedbacks of a few students attending their experiments about the usefulness of their tools. The participants were given some tasks, and were told to follow the task orders so that the conflicts can happen following their predefined scenarios. After the experiments, the students answered the questionnaires containing the Likert scale and free questions. The questions concentrated on how the participants felt about the usefulness of their tools and which was better, with or without the tools supplying the conflict warnings. The conclusions of these experiments are that it is better to have systems supporting the workers in detecting the (potential) conflicts in advance despite their side effects like false warnings.

Because our work is an extension of the mentioned studies, the results of the experiments conducted by these studies demonstrate that our approach in general and CSWMS in specific are also expected to be useful for collaborative software development.

As these types of evaluation are conducted with the simple applications and make the participants follow the predefined situations, they do not cover the aspects concerning the complexity of the real collaborative software development environment, the scale of the project, and the human factors, etc. Hence, developing a mature version of CSWMS, deploying it into practice, and then analyzing its impact on real collaborative software development based on the feedbacks of the developers are necessary to evaluate exactly the effectiveness of our patterns and the proposed approach. In other words, our research is promising but a longitudinal study in the industry is indispensable for a thorough evaluation, which is unfortunately not feasible in our current conditions.

Chapter 8

Related Work

This dissertation is about the inconsistency problem in a collaborative environment. To detect emerging inconsistencies in real time and providing workers much useful information for inconsistency resolution, we have combined the workspace awareness technique with context awareness technique. The change processes are managed to provide the information about the contexts of the changes in the system. In this section, we discuss relevant contributions in the related areas. In addition, we present some existing studies handling the correctness of workflow, especially structure, resource, and temporal verifications, that can be applied to our study for developing a more mature CSWMS.

8.1 Inconsistency Awareness

Most studies on inconsistency in collaborative environments are about conflicts, a type of inconsistency caused by the concurrent changes of different workers to the same artifact or dependency-related artifacts.

Traditional approach uses VCSs [7] in conjunction with the software development environment to address the problem of concurrent access. Workers regularly check-in their changes, and conflicts are detected at the check-in time after workers have finished their work. To catch conflicts earlier, many workspace awareness techniques were proposed to detect conflicts caused by the local changes in the workspaces of the workers before these changes are committed to the remote repository. These techniques can be classified into two categories: pending conflict detection and emerging conflict detection.

8.1.1 Version Control Systems

Version Control Systems are indispensable in collaborative software development because of the ability to maintain different versions of software artifacts that enhances collaboration, parallel development, and global software development. Version control or revision control is the management of changes to a file. Changes are usually identified by a number or letter code, termed the *revision number*. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

Traditional VCSs, for example, Concurrent Version System (CVS) [8] and Subversion [9], use a centralized model where all the revision control functions take place on a shared server. If two developers try to change the same file at the same time, without some

method of managing access, the developers may end up overwriting each other's work. Centralized VCSs solve this problem by file locking or version merging methods.

File locking or pessimistic locking is the simplest method in which only one worker has *write* access to a resource in the central repository at a time. Once one checks-out a file, the others can *read* that file, but not change that file until that worker checks-in the updated version or cancels the checkout. This technique can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file or group of files. However, if the files are locked for too long, other developers may tend to ignore the VCSs and change the files locally, leading to more serious conflicts in future.

Optimistic locking allows multiple workers to edit the same file at the same time. The first worker can always check-in his changes to the central repository successfully. However, a conflict will occur when the second worker checks-in his update if the system could not reconcile the change. In this case, the later check-in worker will need to resolve the conflict by merging the changes, or by selecting one change in favor of the other.

In the modern distributed VCSs (DVCSs), such as Git [10] and Mercurial [11], there is a *master repository* and several local repositories. Each worker makes his own local repository by cloning the master. The workers will commit their changes to the local repository. Local commits only become visible when the workers push their *changeset* to the master. DVCSs keep track of software revisions and allow many workers to work on a given project without necessarily being connected to a common network. However, they lower user awareness of what is being changed in the system. To compensate this, DVCSs provide more advanced support for visioning and merging than centralized VCSs. According to [12], DVCSs like Mercurial have absolutely different storage structure than old centralized VCSs. They save not whole file snapshot, but changes within file which one makes between commits to the local repository.

8.1.2 Workspace Awareness

Detect Pending Conflicts

Crystal [26], which detects conflicts in Mercurial, merges pairs of repositories, including two local repositories of two workers or the local repository of a worker with the remote repository, to detect pending conflicts rather than potential conflicts. The reported conflicts can be textual conflicts, build conflicts, or test conflicts. However, Crystal does not tell the workers which conflicts are exactly happening but one must perform a pull to investigate the conflicts.

Detect Emerging Conflicts on Ongoing Changes

TUKAN [19] is a spatial-model based system integrated in the VisualWorks/ENVY Smalltalk environment. It provides *presence awareness* and conflict awareness by signaling a worker with the presence of other workers and their changes on the artifacts that are related to the artifact on which he focuses. An *artifact-relation model* is built to support conflict awareness. This model is a graph in which software artifacts are represented as nodes and weighted directed relations state semantic dependencies between two nodes. The weight expresses the distance between these two artifacts. The lower the weight of a relation is, the higher the dependency between two connected artifacts is. Change

signals that help prevent direct and indirect conflicts are ranked based on the weight of relations in the artifact-relation model. However, TUKAN only provides the signal when a particular artifact is being focused, so the workers should contact the workers involved or investigate the concurrent changes once they see the signal of interest; otherwise they may forget it.

CollabVS [20] enriches Visual Studio IDE with both conflict notification and communication. It notifies workers of both direct and indirect conflicts involving the artifacts connected by an unlimited dependency path. The workers can choose the granularity of program elements for which dependency checking is done, for instance method, class, or file. A communication section, including chat, audio, or video sections, is shown in-place within the programming environment to help a worker contact the involved workers conveniently. A worker can also set a *watch* on the related artifacts, which are being changed by other workers, to remind them to check for conflicts after some time or after the workers remove focus from the artifacts. However, this may produce side effects on the collaboration of the workers involved since they are compelled to solve the problem instead of solving it willingly.

Palantir [21] monitors the Eclipse workspaces of workers to provide the information about the ongoing changes, and notifies the workers of potential conflicts. Although Palantir informs the workers of conflicts at the code entity level, it captures ongoing changes in the workspaces at the file level. Palantir reports a direct conflict if there are concurrent changes to a single file or an indirect conflict when a file modified by a worker depends on a file that has its public APIs modified by another worker. Palantir adds visual cues on the package explorer to show which files might be conflicting and supplying details of the potential conflict such as the workers involved, location, and severity of conflict in the case of direct conflict. The severity of conflict is measured quite simply by dividing the number of lines that have been added, removed, or changed by the total number of lines in an artifact.

Lighthouse [23] and its extension CASI [24] do not explicitly detect emerging conflicts but prevent them from happening by showing in an *emerging design*, an up-to-date design representation of the code on the workspaces of workers, which source code entities are being changed and by whom. CASI introduces the *Sphere Influence* that shows workers which source code entities are influenced by their changes. Using this information, the workers can avoid a direct conflict by not modifying the same source code entities or avoid a potential indirect conflict by not modifying the entities in the Sphere Influence of other workers.

To reduce false positives, Syde [25] uses a fine-grained change tracking mechanism in which object-oriented systems are modeled as abstract syntax trees (AST), and changes are tree operations. Every time a worker saves a file in his workspace, the change objects are captured and sent to the server. On the server, Syde keeps many ASTs in which each AST represents the latest state of the system in each worker's workspace. Every time a new change operation is received from a client, ASTs of the related workers are compared to detect direct conflicts or syntactic indirect conflicts. The conflicts in Syde are classified into two categories: "yellow" if two workers have inserted, deleted, or changed the same method, and "red" if one of them has committed his change to a VCS. In the current model of Syde, a conflict never involves more than two developers. Therefore, the cost of storing and comparing ASTs is acceptable. However, with more than two workers in the same projects, the current mechanism should be reconsidered.

Similarly to Crystal [26], WeCode [27] computes the presence of merge conflicts, but by continuously integrating all committed changes and uncommitted changes in the workspaces of related workers into a merge workspace shared among them. This merged workspace is then analyzed, compiled, and tested to detect conflicts on behalf of the workers before check-in. WeCode classifies detected conflicts into many types: structural conflicts, language conflicts, behavior conflicts, and test conflicts. Structural conflicts (direct conflicts) are reported when parallel revisions of the same elements cannot be merged; and other conflicts are detected every time the merged workspace is recompiled (language conflicts) and retested (test conflicts) or a conflicts pattern is matched on the merge system.

Like the studies in this category, our research aims to detect emerging inconsistencies by applying the workspace awareness techniques. But unlike them, we consider the inconsistencies at the view of the contexts of changes rather than individual changes. Our tool, CSWMS, which combines the workspace awareness and context awareness techniques, is an extension of these studies. Besides the latest information about the ongoing changes in the workspace of the workers obtained by monitoring their workspaces, we also use the information about the change history and future changes of the ongoing changes supplied by the change processes containing the changes, to detect not only emerging conflicts like these above studies but also other types of inconsistency, and supply the workers with the context of an inconsistency rather than concurrent changes causing a (potential) conflict only, to help them resolve the inconsistency or ignore it in the case of a wrong warning, more easily and accurately.

8.2 Context Awareness

Context awareness is a rather generic concept [29] in which context is the situation within which something exists or happens, and that can help explain it (Cambridge Advanced Learner's Dictionary). Many interpretations of the notion of context have emerged in various fields of research. In this section, we will present some representative related works in the area of context awareness in software development.

Mylyn [31] is a very popular task and application lifecycle management framework for Eclipse. Mylyn provides a Task-Focused Interface for Eclipse to reduce overloaded information and make multi-tasking easy. Users can use Mylyn to define and view their tasks, select a task to work, and identify the resources associated with that task. The context of a task in Mylyn is the interaction of a worker with the system's elements and relations, including all the files, search results, and other relevant information a worker needs to refer to while working on the task. It uses a degree-of-interest (DOI) to represent the relevance of a resource for the task context. Mylyn considers two types of interaction: direct interactions including selection (editor and view selection via mouse or keyboard), edition (textual and graphical edits), and command (operations such as saving, building, preference setting and interest manipulation), and indirect interaction including propagation (interaction propagates to structurally related elements) and prediction (capture of potential future interaction events). The more a user interacts with a resource, the higher DOI value of the resource is, and vice versa. A resource is removed from the task context if its DOI is below a predefined threshold.

[30] proposes a shared awareness model, continuum of relevance (CRI) model, in which

information collected automatically from developer IDE interactions is used to identify explicit orderings of tasks, artifacts, and developers that are relevant to particular work contexts in collaborative software development projects. The CRI model focuses on four interaction types that influence the changing state of a software project: create, update, view, and delete. Each interaction event is also assigned a weight like Mylyn. Although there are many similarities between the approach of Mylyn and [30], Mylyn does not currently focus on distributed software development but its main objective is to reduce information overload and make multi-tasking easier for an individual, while the model of [30] aims to enhance the coordination among workers.

There are some similarities between Mylyn and CSWMS where a worker must also define a CSW, start his assigned activity, and finish it. A CSW can be transformed into a sequence of Mylyn tasks and vice versa. However, the context concept of our research is different from of Mylyn. We consider the CSW of a change as the context of the change. Also, Mylyn does not pay attention to the inconsistency problem. 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.

Regarding [30], by enhancing coordination among worker, its proposed CRI model may be helpful for our future work relating to automatic generation of CSWs.

8.3 Process Centered Software Development Environment

In a process centered software development environment, workers follow a sequence of steps given by predefined structures. The environment manages the assignment of tasks to workers, monitors their execution, and invokes appropriate tools. By allowing a worker to follow a CSW for implementing a change request, our approach is close to the approach of process support tools. However, because the change process concept considered in our research is different from business processes that are considered by most process centered environments but is close to the cooperative process concept given in [33], we introduce only [33] in this section.

[33] presents a set of requirements for a workflow management system that aims to support cooperative workflows, specially the requirements for high flexibility and dynamicity. These requirements are implemented by a Bonita workflow management system. Besides providing the same kind of support as a traditional workflow management system (WFMS) for business processes, Bonita supports the cooperative processes that can be created, executed, and modified during its execution easily and dynamically. In Bonita, processes can be executed flexibly due to the ability to start an activity in advance, namely anticipation. The main idea is that an activity can be executed even when all its activation conditions are not met at the beginning but at some time before the termination of the activity. Therefore, besides common states of an activity in a WFMS such as initial, executable, executing, aborted, and terminated, an activity in Bonita can be in the state of anticipable or anticipating canceled. An activity can be executed as soon as it is created. It is then in the state Executable.

Although there are many similarities between CSW and the cooperative process defined in [33], especially the need for dynamic change during execution, CSW extends the *cooperative process* with information about artifacts and atomic changes. CSWMS is also

distinguished from Bonita model because we consider a specific situation, change process in collaborative environment, but Bonita model considers cooperative process in general. Also, inconsistency awareness is not mentioned in Bonita but a key issue of CSWMS.

8.4 Workflow Correctness

In this dissertation, we pay attention to the data aspect of CSWs to detect (potential) data-related inconsistencies. Because verifying other aspects of CSWs, such as control structure, resource constraints, and temporal constraints, is important to the development of a mature version of CSWMS, this section describes some related studies that can be useful for further development of CSWMS.

Structure verification verifies the consistency of a workflow's structure, for example, deadlock, livelock, and lack of synchronization. Temporal verification aims to verify the temporal consistency of a workflow specification. Resource verification verifies the existences of any resource conflicts among activities in a workflow. Existing studies solved these problems separately or together.

To analyze the properties of a workflow, it must be able to specify the workflow and its expected behaviors using a formalism with well-defined semantics. Among the popular formalisms in this area, Petri-net may be the most frequently used process modeling technique in workflow verification. Graph theory, logic, and set theory are also used rather widely.

[57] used directed graphs to specify a workflow and then analyzed the generated directed graph to detect deadlocks or lack of synchronizations. In [58], temporal logic was employed to describe workflows and the corresponding logic expressions were analyzed to verify the structural correctness of a workflow specification. [43, 59] modeled workflows as Petri-nets where activities are mapped by transitions, and dependencies between activities are mapped by places and arcs. Then, the structural correctness of a workflow specification could be verified through analyzing the corresponding Petri-net.

Regarding temporal verification, [60] defined a timed workflow graph in which each activity node is augmented with the earliest end time and the latest end time. Temporal constraints can be calculated using the modified critical path method at build-time and process instantiation time, and then enforced at run-time. [61] assigned a time interval to each workflow activity as duration constraints and verified temporal requirements and inconsistencies of workflow systems. [62] proposed a timed workflow process model incorporating the time constraints, the duration of activities, the duration of flows, and the activity distribution with respect to the multiple time axes, into the conventional workflow processes. The model provided an approach to temporal consistency checking during both build-time and run-time. In [63], temporal constraint Petri-nets were employed to specify workflows, and then the temporal feasibility of a workflow at build-time was tested. [64] proposed a timing constraint workflow net (TCWF-net), extending WF-Net [43] with time information, to specify the timing constraints. A directed network graph (DNG) based workflow model is converted into a TCWF-net first. Then schedulability and boundedness of the TCWF-net are analyzed. [65] applied Time Petri-nets to model the temporal behavior of workflow systems using TINA as a tool to support the verification of the activities deadlines.

[44] discussed the analysis of resource constraints of a workflow specification and pre-

sented an approach with corresponding algorithms to the resource consistency of the workflow specification. [66] introduced Resource-Constrained Workflow Nets (RCWF-nets) and adapted the soundness from WF-nets for RCWF-nets. The analysis method is based on flow structure and only one resource type assumption, and it deals with the instances of the same workflow model. [45] defined the Time Constraint Workflow Net and mapped the workflow concepts into this net to model workflows. Then, the resource constraints on concurrent workflows were analyzed. [46] used hybrid automata to model the influences between concurrent workflows, and adopted a model checking technique to detect resource conflicts. [47] proposed RND_WF_Net to model a kind of workflow constrained by resources and non-determined duration. A verification approach and resolution strategies for resource conflicts between activities were also presented.

In short, we have presented some existing approaches to the workflow correctness problem. These studies are good references when the workflow correctness problem is considered more completely in the mature version of CSWMS.

8.5 Summary

We have provided an overview of the existing works relating to our research. Our inconsistency awareness using both workspace awareness technique and context awareness technique can recognize many types of inconsistency rather than conflicts only. In addition, our approach can provide much more information about the context of an inconsistency compared with the existing works. We also discussed related works in the field of workflow verification, including structure verification, temporal verification, and resource verification, that can be employed in further development of CSWMS.

Chapter 9

Conclusion

In collaborative software development, many change requests, such as adding or modifying a feature, or fixing bugs, can be implemented by different workers within a time interval. Each worker conducts his own *change process* that is a sequence of tasks applying changes to a set of artifacts to fulfill a change request. However, because of communication problems and the complex and changeable nature of software, the workers could not have enough information about the work of the others. These problems are more serious in current practices with parallel development and distributed environment. Therefore, a change of a worker may unexpectedly affect the changes of the others. As a result of that, inconsistencies originated from the affected artifacts may happen.

In this dissertation, we take into account the problems not addressed by the existing studies in the area of inconsistency awareness. Differently from these studies that concentrated only on concurrent changes and considered them separately, we pay attention to 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. By considering the inconsistency problem under the view of the change processes containing the changes, we have first identified the patterns of inconsistency that are not mentioned in the previous studies. Then we have proposed an environment, where the change processes are represented explicitly and are managed with regard to the patterns of inconsistency. In order to detect these inconsistencies in advance, we have collected the latest information about the ongoing changes at the workspaces of the workers and the progress of the change processes to notify the workers in advance emerging inconsistencies along with their contexts. The context of an inconsistency, including the contents of the changes causing the inconsistency and the change processes of the changes, can help the workers resolve the inconsistency easily.

In summary, we have developed the model and the tool toward a Change Support Environment (CSE) able to detect in advance the (potential) inconsistencies that previous works can not detect, and to supply the contexts of these inconsistencies, rather than just the involved changes themselves like the previous works. Therefore, our work can contribute to building a safer and more efficient collaborative software development environment.

9.1 Contributions

In this dissertation, we have made the following contributions.

1. We have defined the patterns of inconsistency classified based on the relationships between the changed artifacts, the time orders of the tasks applying the changes to the artifacts, and the change processes of the changes.

2. We have proposed a context-based approach to solve the inconsistency problem more effectively. The change processes in collaborative software development are managed and are used to provide the contexts of the changes in the system. Our inconsistency awareness technique combines monitoring the workspaces of the workers (workspace awareness) with managing the progress of the change processes (context awareness) to detect in advance (potential) inconsistencies in real time. Information about the changes causing an inconsistency and their change processes, namely the context of the inconsistency, is provided to help the workers fully comprehend the inconsistency before resolving it.

3. Based on the proposed approach, we have developed a Change Support Workflow Management System that allows the workers to define, execute, and modify their change processes easily, and to receive inconsistency warnings along with the contexts of the inconsistencies to resolve the inconsistencies in advance.

4. We have also given a formal method for modeling the main behaviors of CSE in CP-nets. CPN Tools is used in verifying the generated model to detect the patterns of inconsistency. Our method can be applied to model and verify the data-flows of other types of workflows.

These achievements have demonstrated the feasibility of our proposed approach and open up chances to apply it into real software development environments.

9.2 Limitations and Future Work

In this section, we present potential future work and discuss the shortcomings of our work, from which some of the future work is originated.

1. **CSW Generation Support.** In the current prototype of CSWMS, workers must define a CSW representing a change process by themselves. Although we have minimized the compulsory information a worker must provide (names of activities and their orders), defining CSWs is still a barrier to evaluating the effectiveness of our approach, because in simple laboratory examples, it is difficult for the users to compare the cost of defining CSWs with the benefits of defining CSWs, such as having a plan to follow, reviewing their change history easily, understanding the works of other workers more exactly without contacting them, and resolving inconsistency more easily. Also, forcing the users to change their usual coding behaviors, for instance defining a CSW before implementing a change request, and starting and stopping an activity, can make them have a bias in favor of not defining CSWs.

   To deal with this problem, we are considering two following possibilities:

   - **Generating CSWs automatically by extracting the necessary information for reconstructing CSWs from the repositories of VCSs.** We

can try to automatically generate CSWs in which each activity corresponds to a commit of a worker. If the workers are working on different branches sharing the same root, each branch will be considered as a CSW. In the case the workers are working on the same branch, we can automatically generate a CSW of a worker by connecting his current changes, not yet committed, with his committed changes. Some heuristics can be considered such as author, same committer, time, within a time interval, or keyword. Mining the compact set of changes committed to the repository, we can suggest a worker the likely changes based on the coupling between the likely changes and his ongoing changes in the past. To handle the new features, we need to extend the CSWMS with some new components. The **System State Management** component helps create, update, and manage access to Abstract Syntax Tree (AST) representing the newest state of the system in VCS. The **CSW Generation** component generates CSWs associated with the inconsistency-involved changes using the commit history and current changes of the workers. Finally, the **VCS Handler** component supports other components in accessing VCS and notifies **System State Management** and **Inconsistency Analysis** of the arrival of a new commit.

- **Developing CSWMS based on Mylyn [31] instead of the dynamic WFMS Bonita [33].** Mylyn is the most popular IDE tool for task and application lifecycle management (ALM) on Eclipse. The wide acceptance of Mylyn promises that Mylyn users will not mind defining a CSW describing their change progresses. A CSW can be mapped to a task and each activity in a CSW to a subtask of Mylyn. Using Mylyn, Workspace Wrapper can analyze the interaction events collected by Mylyn to predict likely changes following the ongoing changes. Mylyn classifies interaction events into five types: *selection*, *edit*, *command*, *propagation*, and *prediction*, among which the *selection* and *edit* events are useful for inconsistency analysis. Artifacts associated with *edit* event can be considered as ***w**riten* artifacts. Artifacts associated with the *selection* event can be referred to as potential ***r**ead* artifacts. Possibility to become a ***r**ead* artifact is proportional to the frequency with which artifacts appear in *selection* events.

2. **Mining the bug repository and the source code repository of VCSs to find the patterns of inconsistency.** Our research does not provide empirical studies for proving the popularity of our patterns of inconsistency, because this work requires a lot of time and belongs to other research areas with some specific techniques, and hence it is difficult to be solved in the scope of this dissertation. However, this research direction will strengthen the current research, especially by discovering new patterns of inconsistency or specific cases of the patterns of inconsistency to help our system improve the accuracy of inconsistency detection.

3. **Handling model artifacts**

   The current prototype of CSWMS focuses on artifacts that are source code elements because of their importance as a fully executable description of a software system, and the widely supports of IDEs and VCSs. Since source code is the output of the coding phase where developers translate model artifacts, output of the design

phase in Software Development Life Cycle, into executable programming language code, customizing the current system to handle not only source code but also model artifacts will increase its usefulness and practicability. The method given by [39] will be considered for automatically generating dependency relationships between UML elements.

4. **Extending the current system for maintaining purpose.** In addition to handling the inconsistencies, by managing the change processes, CSWMS can be improved to help maintain and update software systems and increase the reusability of the change processes. This can satisfy the questions about the overhead in setting up and maintaining the change processes for the improved early detection and resolution of (potential) inconsistencies.

9.3 Closing Words

In this dissertation, we have shown that managing change processes in a collaborative software development environment can help deal with the inconsistency problem more effectively. By supporting workers in planning their changes and collecting and analyzing information about their change schedules, their change history, and their ongoing changes, to prevent, detect, and resolve an inconsistency proactively, our research contributes to building a safer and more effective collaborative software development environment.

Publications

[1] PHAN Thi Thanh Huyen and Koichiro Ochimizu, "An Inconsistency Management Support System for Collaborative Software Development", IEICE Transactions on Information and Systems (accepted).

[2] PHAN Thi Thanh Huyen, Kunihiko Hiraishi, and Koichiro Ochimizu, "Modeling and Verification of Change Processes in Collaborative Software Engineering", Proc. of the Workshops of the 13th International Conference on Computational Science and Applications (SEPA 2013), LNCS, vol. 7973, pp. 17-32, Hochiminh, Vietnam, June, 2013.

[3] Kazuhiro Ogata and PHAN Thi Thanh Huyen, "Specification and Model Checking of the Chandy & Lamport Distributed Snapshot Algorithm in Rewriting Logic", Proc. of the 14th International Conference on Formal Engineering Methods (ICFEM 2012), LNCS, vol. 7635, pp. 87-102, Kyoto, Japan, October, 2012.

[4] PHAN Thi Thanh Huyen and Koichiro Ochimizu, "Toward Inconsistency Awareness in Collaborative Software Development", Proc. of the 18th Asia Pacific Software Engineering Conference (APSEC 2011), IEEE, pp. 154-162, Hochiminh, Vietnam, December, 2011.

[5] PHAN Thi Thanh Huyen, Kunihiko Hiraishi, and Koichiro Ochimizu, "A Workflow-based Change Support Model for Collaborative Software Development", Proc. of the IEICE Society Conference, pp. S-23 - S-24, 2011.

[6] PHAN Thi Thanh Huyen and Koichiro Ochimizu, "A Change Support Model for Distributed Collaborative Work", JAIST Technical Report IS-RR-2011-003, pp. 1-10, July, 2011.

[7] Koichiro Ochimizu, PHAN Thi Thanh Huyen, Masayuki Kotani, Saw Sand Aye, "Towards Software Development Environments for Distributed Cooperative Works", Proc. of Japan-Vietnam Workshop on Software Verification: From Mathematical Logic and Formal Language Theory to Software Engineering (JVSE10), 2010.

[8] PHAN Thi Thanh Huyen and Koichiro Ochimizu, "Detecting and Repairing Unintentional Change in In-use Data in Concurrent Workflow Management System", Proc. of the Workshops of the 31st International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (PETRI NETS 2010), pp. 331-351, Braga, Portugal, June, 2010.

[9] PHAN Thi Thanh Huyen and Koichiro Ochimizu, "Detection of Unintentional Change on In-use Data for Concurrent Workflows", Proc. of the 8th International

Conference on Software Engineering Research and Practice (SERP'10), pp 277-283, Nevada, USA, July, 2010.

Bibliography

[1] K. Ochimizu, H. Murakosi, K. Fujieda, and M. Fujita, “Sharing Instability of a Distributed Cooperative Work”, Proc. of the international symposium on the principles of software evolution (ISPSE 2000), IEEE Computer Society Press, pp. 36-45, 2000.

[2] J. Whitehead, “Collaboration in Software Engineering: A Roadmap”, Proc. of the 2007 Future of Software Engineering (FOSE 2007), IEEE Computer Society Press, pp. 214-225, 2007.

[3] H. Murakosi, A.Shimazu, and K. Ochimizu, “Construction of Deliberation Structures in E-mail Communications”, Computational Intelligence, vol.16, no.4, pp. 570-577, 2000.

[4] J. Rumbaugh, I. Jacobson, and G. Booth, “The Unified Modeling Language Reference Manual”, Addison-Wesley, ISBN 0-201-30998-X, 1999.

[5] K. Ochimizu, “Software Process Model for Evolutionary Software Development - Coordination Support for a Distributed Cooperative Software Development”, Computer Software, vol. 15, no. 4, pp 73-77, 1998 (In Japanese).

[6] J. D. Herbsleb and A. Mockus, “An Empirical Study of Speed and Communication in Globally Distributed Software Development”, IEEE Transactions on Software Engineering, vol.29, no. 6, pp. 481-494, 2003.

[7] K. Altmanninger, M. Seidl, and M. Wimmer, “A Survey on Model Versioning Approach”, International Journal of Web Information Systems, vol. 5, iss. 3, pp. 271-304, 2009.

[8] Concurrent Versions System, http://www.nongnu.org/cvs/

[9] Subversion, http://subversion.tigris.org/

[10] Git, http://git-scm.com/

[11] Mercurial, http://mercurial.selenic.com/

[12] ”Mercurial: The Definitive Guide”, http://hgbook.red-bean.com/

[13] Adaptable Model Versioning, http://modelversioning.org/

[14] SMoVer, www.smover.tk.uni-linz.ac.at/

[15] I. Barone, A.D. Lucia, F. Fasano, E. Rullo, G. Scanniello, and G. Tortora, "CO-MOVER: Concurrent Model Versioning", Proc. of the IEEE International Conference on Software Maintenance, IEEE Computer Society Press, pp. 462-463, 2008.

[16] G. Fitzpatrick, P. Marshall, and A. Phillips, "CVS Integration with Notification and Chat: Lightweight Software Team Collaboration", Proc. of the 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW'06), pp. 49-58, 2006.

[17] Borland Together, http://www.borland.com/us/products/together/

[18] IBM's Jazz.net platform, http://jazz.net/

[19] T. Schummer and J. M. Haake, "Supporting Distributed Software Development by Modes of Collaboration". Proc. of the 7th European Conference on Computer Supported Cooperative Work, pp. 79-98, 2001.

[20] P. Dewan and R. Hegde, "Semi-synchronous Conflict Detection and Resolution in Asynchronous Software Development", Proc. of the 10th European Conference on Computer-Supported Cooperative Work, Springer, pp. 159-178, 2007.

[21] A. Sarma, G. Bortis, and A. van der Hoek, "Towards Supporting Awareness of Indirect Conflicts across Software Configuration Management Workspaces," Proc. of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 94-103, 2007.

[22] L. Hattori and M. Lanza, "An Environment for Synchronous Software Development", Proc. of the 31st ACM/IEEE International Conference on Software Engineering - New Ideas and Emerging Results Track (ICSE 2009), IEEE Computer Society Press, pp. 223-226, 2009.

[23] I. da Silva, P. Chen, C. V. der Westhuizen, R. Ripley, and A. van der Hoek, "Lighthouse: Coordination through Emerging Design", Proc. of OOPSLA Workshop on Eclipse Technology eXchange (ETX 2006), ACM Press, pp. 11-15, 2006.

[24] F. Servant, J.A. Jones, and A.V.D. Hoek, "CASI: Preventing Indirect Conflicts through a Live Visualization", Proc. of the Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '10), pp. 39-46, 2010.

[25] L. Hattori and M. Lanza, "Syde: A Tool for Collaborative Software Development", Proc. of the 2010 International Conference on Software Engineering (ICSE'10), pp. 235-238, 2010.

[26] Y. Brun, R. Holmes, M.l Ernst, and D. Notkin, "Proactive Detection of Collaboration Conflicts", Proc. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering of ESEC/FSE, pp. 168-178, 2011.

[27] M. L. Guimaraes and A. R. Silva, "Improve Early Detection of Software Merge Conflicts", Proc. of the 2012 International Conference on Software Engineering (ICSE 2012), pp. 342-352, 2012.

[28] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Fine-grained Management of Software Artefacts: the ADAMS System", Software Practice and Experience, vol. 40, no. 11, pp. 1007-1034, 2010.

[29] I. Omoronyia1, J. Ferguson, M. Roper, and M. Wood, "A Review of Awareness in Distributed Collaborative Software Engineering", Software Practice and Experience, vol. 40, pp. 1107-1133, 2010

[30] I. Omoronyia1, J. Ferguson, M. Roper, and M. Wood, "Using Developer Activity Data to Enhance Awareness during Collaborative Software Development", Journal Computer Supported Cooperative Work archive, vol.18, iss. 5-6, pp. 509-558, 2009.

[31] M. Kersten and G.C. Murphy, "Using Task Context to Improve Programmer Productivity", Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1-11, 2006.

[32] R. Robbes and M. Lanza, "Spyware: A Change-Aware Development Toolset", Proc. of the 2008 International Conference on Software Engineering (ICSE 2008), ACM Press, pp. 847-850, 2008.

[33] F.C. Charoy, A. Guabtni, and M.V. Faura, "A Dynamic Workflow Management System for Coordination of Cooperative Activities", Proc. of Business Process Management (DPM '06), pp. 205-216, 2006.

[34] JGraph, http://www.jgraph.com

[35] Dependency Finder, http://depfind.sourceforge.net/

[36] S.A. Bohner and R.S. Arnold, "Software Change Impact Analysis", IEEE Computer Society Press, ISBN 0-818-67384-2, 1996.

[37] T. Zimmermann, "Mining Workspace Updates in CVS", Proc. of the Fourth International Workshop on Mining Software Repositories (MSR'07), 2007.

[38] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley, ISBN 0-201-65793-7, 2000.

[39] M. Kotani and K. Ochimizu, "Automatic Generation of Dependency Relationships between UML Elements for Change Impact Analysis", Journal of Information Processing Society of Japan, vol.49, no.7, pp 2265-2291, 2008 (In Japanese).

[40] PatternWeaver, http://pw.tech-arts.co.jp/

[41] K. Jensen and L.M. Kristensen, "Colored Petri Nets - Modeling and Validation of Concurrent Systems", Springer, 2009.

[42] CPN Tools, http://cpntools.org/

[43] W.v.d Aalst and K.M.v. Hee, "Workflow Management: Models, Methods, and Systems", MIT press, Cambridge, MA, 2004.

[44] H.Li, Y. Yang, and T.Y. Chen, "Resource Constraints Analysis of Workflow Specifications", Journal of Systems and Software, vol.73, iss.2, pp. 271-285, 2004.

[45] J. Zhong and B. Song, "Verification of Resource Constraints for Concurrent Workflows", Proc. of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SNASC 2005), pp. 253-261, 2005.

[46] S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama, "Constraint Verification for Concurrent System Management Workflows Sharing Resources", Proc. of the Third International Conference on Autonomic and Autonomous Systems (ICAS07), 2007.

[47] Q. Zeng, H. Wang, D. Xu,H. Duan, and Y. Han, "Conflict Detection and Resolution for Workflows Constrained by Resources and Non-determined Duration", Journal of Systems and Software, vol.81, iss.9, pp 1491-1504, 2008.

[48] S. Sadiq, M. Orlowska, and W. Sadiq, "Data Flow and Validation in Workflow Modeling", Proc. of the 15th Australasian Database Conference (ADC '04), pp. 207-214, 2004.

[49] S.X. Sun, J.L. Zhao, J.F. Nunamaker, "Formulating the Data Flow Perspective for Business Process Management", Information Systems Research, vol. 17, no. 4, pp 374–391, 2006.

[50] S. Fan, W.C. Dou, and J. Chen, "Dual Workflow Nets: Mixed Control/Data-Flow Representation for Workflow Modeling and Verification", Proc. of Advances in Web and Network Technologies and Information Management (APWeb/WAIM 2007 Workshops), LNCS, vol. 4537, pp 433-444, Springer-Verlag, Berlin, 2007.

[51] H.S. Meda, A.K. Sen, and A. Bagchi, "Detecting Data Flow Errors in Work-flows: A Systematic Graph Traversal Approach", Proc. of the 17th Workshop on Information Technology & Systems (WITS-2007), pp. 133-138, 2007.

[52] A. Awad, G. Decker, and N. Lohmann, "Diagnosing and Repairing Data Anomalies in Process Models", Proc. of International Workshop on Business Process Design, LNBIP, pp 1-24. Springer, Heidelberg, 2009.

[53] N. Trcka, W.M.P. van der Aalst, and N. Sidorova, "Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows", Proc. of the 21st International Conference on Advanced Information Systems (CAiSE'09), LNCS, vol. 5565, pp. 425-439, Springer-Verlag Berlin Heidelberg, 2009.

[54] W.v.d Aalst and C. Stahl, "Modeling Business Processes: A Petri net-Oriented Approach", The MIT Press, 2011.

[55] N. Sidorova, W.M.P. van der Aalst, and N. Trcka, "Soundness Verification for Conceptual Workflow Nets with Data: Early Detection of Errors with the Most Precision Possible", Information Systems, vol. 37, no. 7, pp. 1026-1043, Springer-Verlag Berlin Heidelberg, 2009.

[56] Emanuel Giger, Martin Pinzger, and Harald C. Gall, "Can We Predict Types of Code Changes? An Empirical Analysis", Proc. of Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, pp. 217-226, 2012.

[57] W. Sadiq and M.E. Orlowska,"Analyzing Process Models Using Graph Reduction Techniques", Information System, vol. 25, iss. 2, pp. 117-134, 2000.

[58] G. Greco, A. Guzzo, and D. Sacca, "A Logic-based Formalism to Model and Analyze Workflow Executions", Proc. of CEUR Workshop Proceedings, vol.74, 2003.

[59] J. Lee and L.F. Lai, "A High-level Petri-nets-based Approach to Verifying Task Structures", IEEE Transactions on Knowledge and Data Engineering, vol. 14 no. 2, pp. 316-335, 2002.

[60] J. Eder, E. Panagos, and M. Rabinovich, "Time Constraints in Workflow Systems", Lecture Notes in Computer Science", vol. 1626, pp. 286-300, 2000.

[61] O. Marjanovic, "Dynamic Verification of Temporal Constraints in Production Workflows", Proc. of the AustralasianDatabase Conference, pp. 74-81, 2000.

[62] H. Zhuge, T.Y. Cheung, and H.K. Pung, "A Timed Workflow Process Model", Journal of Systems and Software, vol. 55, iss.3, pp 231-243, 2001.

[63] N.R. Adam, V. Atluri, and W.K. Huang, "Modeling and Analysis of Workflows Using Petri nets", Journal of Intelligent Information Systems, vol. 10, iss.2, pp. 131-158, 1998.

[64] J. Li, Y. Fan, and M. Zhou, "Timing Constraint Workflow Nets for Workflow Analysis", IEEE Transactions on Systems, Man, and Cybernetics, Part A, vol. 33 iss. 2, pp. 179-193, 2003.

[65] Pedro M. Gonzalez del Foyo and Jose Reinaldo Silva,"Using time petri nets for modeling and verification of Timed constrained workflow systems", ABCM Symposium Series in Mechatronics, vol. 3, pp. 471-478, 2008.

[66] K.M van Hee, A. Serebrenik, N. Sidorova, and M. Voorhoeve, "Soundness of Resource-constrained Workflow Nets", Proc. of ICATPN, Lecture Notes in Computer Science, vol. 3536. Springer, pp. 250-267, 2005.