Prevent Collaboration Conflicts with Fine Grained Pessimistic Locking

Martin Eyl[1], Clemens Reichmann[1] and Klaus D. Müller-Glaser[2]
[1]*Vector Informatik GmbH, Ingersheimer Straße 24, 70499 Stuttgart, Germany*
[2]*Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany*

Keywords: Fine Grained Software Configuration Management System, Abstract Syntax Tree, Pessimistic Locking.

Abstract: There are two main strategies to support the collaboration of software team members working concurrently on the same source code: pessimistic locking and optimistic locking. Optimistic locking is used far more often because a pessimistic lock on the smallest unit stored in the Software Configuration Management (SCM), which is a usually a file, often causes conflict situations, where a developer wants to change the already locked code. Optimistic locking can cause direct and indirect merge conflicts which are costly to resolve and affect productivity. The novelty of our approach is to define a meta-model for the source code (Abstract Syntax Tree) and use pessimistic locking on model artefacts and therefore allow parallel editing of the same class or even method but still preventing direct and indirect merge conflicts. Thereby the developer keeps an isolated workspace and the developer decides when to commit the finished source code. This paper introduces a concept for this solution and a prototype based on Eclipse.

1 INTRODUCTION

In many software development projects the increasing number of requested features and their complexity makes it necessary to have an increasing number of software developers working in parallel on the same source code (Perry, 2001; Estublier, 2005). Therefore a solution is needed for parallel editing of source code. Different Software Configuration Management Systems (SCM) address the problem of conflicting source code changes and provide appropriate solutions which comes with benefits and weaknesses (Conradi, 1998; Grinter, 1995).

1.1 Pessimistic and Optimistic Locking

There are two main approaches: pessimistic and optimistic locking (Sarma, 2003; Levin, 2013). In the optimistic approach parallel changes of the same source code lines are allowed. Any conflicts must be solved before the source code can be committed into SCM repository. In some cases these conflicts are trivial to resolve. However in other cases it is very complicated and error-prone to merge the source code (Sarma, 2007; Dewan, 2008).

In the pessimistic approach the software developer needs a lock for the source code before it can be modified. The lock is exclusive which means that other developers cannot retrieve the lock for the source code until it is released. No concurrent changes of the same source code are possible. The smallest unit stored in an SCM repository is typically a file (Broschy, 2009). Pessimistic locking on a file is too restrictive and causes too often conflict situations, where a developer wants to change a file that is already locked.

1.2 Direct and Indirect Conflicts

There are two different classes of conflicts: direct conflicts and indirect conflicts (Brun, 2011). A direct conflict is caused by changing the same line of source code by two developers in their local workspaces at the same time. The SCM repository can detect such conflicts during the commit of source code.

An indirect conflict originates from changing the source code in a file which affects concurrent changes of a second developer in the same or another source code file. If indirect conflicts are not detected and resolved then they can cause syntax errors in the source code stored in the SCM

Eyl M., Reichmann C. and MÃijller-Glaser K.
Prevent Collaboration Conflicts with Fine Grained Pessimistic Locking.
DOI: 10.5220/0006119703120319
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 312-319
ISBN: 978-989-758-210-3
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

repository. Also new defects can arise which only exists in the combination of both source code changes. Continuous Integration (CI) is used to detect the syntax errors. The new defects can only be found via tests for example via automated tests during CI (Eyl, 2016). A typically example of an indirect conflict is the following: one developer renames a method in one class and the other developer adds a new invocation of this method in a second class with the original name. If both developers commit the source code at the same time into the SCM repository then there is no direct conflict but there is a syntax error in the second class calling a method which does not exists with this name anymore.

1.3 Motivation and Objectives

As already mentioned, direct conflicts can be very difficult and costly to solve (Estublier, 2005). If there are several lines of source code involved for example in an algorithm then in some cases it is not possible to solve the conflict without the help of the developers who did the conflicting changes (Grinter, 1995). Sometimes it is easier to undo your own changes and to redo the changes in the updated source code. These merge conflicts are very annoying because it prevents the developer for committing the source code although his/her work is already finished. Indirect conflicts which cause syntax errors in the SCM repository can block other team members (which CI cannot prevent). If the developers update their source code with the source code from the SCM repository containing a syntax error, the developers have to fix the syntax error before they can continue to work. Also a CI run with automated tests cannot be executed if there are syntax errors in the code.

By using pessimistic locking it is never necessary to merge source code and therefore the developer can commit the source code anytime. If the developer wants to change already locked source code it is necessary to communicate with the colleague to coordinate the work. Pessimistic locking can also be used to prevent indirect conflicts which cause syntax errors in the SCM repository. However pessimistic locking of a complete file is difficult because only one developer can change the code in the file. Pessimistic locking is practical when a fine grained pessimistic locking is supported which allows parallel editing of a class or even a method.

We implemented a prototype with the following objectives:

- Defining a meta-model for the source code (Abstract Syntax Tree).
- Pessimistic locking on one or more model artefacts.
- Allow parallel editing of classes and methods by using only a small number of fine grained locks.
- No syntax errors in the SCM repository.
- Minimal automatic locking during editing of the source code in the editor.
- Keep isolation from other developers and the developer decides when to commit the source code.

Using optimistic locking a direct merge conflict can only be resolved after an update of the affected file. Pessimistic locking forces sequential changes of the source code. Therefore an update is necessary when the developer acquires a lock for an AST artefact. The developer is forced to update earlier than with optimistic locking. However the developer can freely decide when to commit the finished source code.

The prototype has been implemented for Java and is an extension of Eclipse Integrated Development Environment (IDE) (Eclipse, 2016). The fine grained pessimistic locking is part of a larger research project which is called Morpheus.

2 ABSTRACT SYNTAX TREE (AST)

The AST provides us with the means to find dependencies in the code which we need to set the correct locks to prevent indirect conflicts. Also the AST is fine grained enough for the fine grained pessimistic locks. An AST is composed of AST nodes and relations between the nodes. There are two types of relationships: composition and association.

AST nodes are for example *Package Declaration*, *Type Declaration*, *Method Declaration*, *Method Invocation*, *If Statement* and so forth. In this paper all AST nodes are written in italic. The AST nodes build up a hierarchically tree via the composition for example the *Method Declaration* is contained in the *Type Declaration*. We call the *Type Declaration* “composite node” and the *Method Declaration* “component node”.

The association is a usage relation and comes often with a declaration and a use of the declared artefact. We call the declaration “supplier node” and the other side of the relation “client node”. The client depends on the supplier. The invocation of a

method is such an association between a *Method Invocation* and a *Method Declaration*. Another example is the inheritance of a class from another class which is an association between two *Type Declarations*.

When the developer wants to change source code, the corresponding AST nodes have to be locked to prevent parallel modifications. To prevent indirect conflicts depended artefacts also have to be considered. Therefor we have to define some rules (see Section 4).

2.1 The AST as Meta-Model

AST nodes have to be locked and it is difficult to keep track of the locks when the AST node does not have a unique object identifier (OID) especially the AST nodes without a name for example a "for loop" or an "if statement". For this reason we used the Meta Data Framework (MDF) of PREEvision (Vector, 2016; Zhang 2011). PREEvision is a model based, 3-tier application used mainly in the automobile industry. MDF is based on the OMG's Meta Object Facility (MOF) Standard (OMG, 2016). MDF allows us to define a meta-model for the AST and to generate the Java source code for the model. PREEvision also provides functionality for storing the model into a data backbone (PREEvision server). We use PREEvision with the AST meta-model as a model based Software Configuration Management (SCM) repository.

The AST model can be edited by using a structured editor, for example the projectional editor of JetBrains Meta Programming System (MPS) (JetBrains, 2016). Alternatively an existing Java text editor has to be extended so that the editor is aware of the AST artefacts. We chose the second solution and added the required functionality to the Java text editor of the Eclipse Integrated Development Environment (IDE). We call this text editor Java AST Editor. The editor makes sure that changing and refactoring of the source code (for example rename or move) does not delete and recreate AST nodes but only changes the existing AST nodes. For example if the developer changes the name of a method then the *Method Declaration* is not deleted and recreated. Instead the name of the *Method Declaration* is modified.

2.2 Meta-Model and Syntax Errors

In the source code text the link between Method Declaration and Method Invocation is realized via full qualified name: package name, class name and the method name. By changing the name in the method declaration but not in the method call the link is broken because the names no longer match and we have caused a syntax error. In the meta-model the link between *Method Declaration* and *Method Invocation* is represented as an association with the cardinality "1": the *Method Invocation* has to have exactly one *Method Declaration*.

So, the meta-model makes sure that the syntax error "Method Invocation without Method Declaration" is not possible. Just by using a meta-model some syntax errors can no longer occur. However the meta-model does not prevent all syntax errors. For example if the number of parameters or the type of the parameters is not correct in the method invocation then it is still possible to build up a valid AST which can be stored in the server. The Java compiler detects such syntax errors in the source code and it is not reasonable to add all this functionality to the meta-model. To make sure that no syntax errors can occur in the SCM repository we have to ensure two things: Firstly, the developer should not be able to commit source code with syntax errors into the SCM repository. If Eclipse indicates any syntax errors in the current source code then the commit will be rejected. Secondly, syntax errors because of indirect conflicts have to be prevented with pessimistic locking.

3 FINE GRAINED AST LOCKS

An exclusive lock is a marker for an AST node which determines who can currently modify the AST node. A lock can be acquired by sending a request to the server where a list of all locks is stored. For each lock the server stores the object identifier (OID) of the AST node, the user who owns the lock and the date when the lock has been acquired. The locks are automatically released after the commit of the AST into the server.

Our objective is to allow as much parallel editing as possible. In order to achieve this, we need an additional lock type. For example if the developer wants to add new source code which contains a new method call then a new AST node (*Method Invocation*) is created and an association to the according *Method Declaration*. To ensure that another developer is not modifying the method signature or deleting the method at the same time the developer needs an exclusive lock for the *Method Declaration*. This exclusive lock would prevent other developers to add also a new method call to this method because only one developer can own an

SCIENCE AND TECHNOLOGY PUBLICATIONS

exclusive lock for an AST node. However this parallel editing would never cause a direct or indirect conflict. The problem can be solved with a shared lock. Several developers can own a shared lock for the same AST node. No exclusive lock can be acquired for an AST node with a shared lock. The shared lock ensures that other developers do not change the method signature but still allow several developers to call this method at the same time.

4 LOCK RULES

In this section we want to clarify which Abstract Syntax Tree (AST) nodes have to be locked with which lock type (shared or exclusive) when changing the source code to prevent any merge conflicts. Therefore we will define several rules.

We have to consider the following types of conflicts: direct conflict, indirect conflict and name conflict.

4.1 Direct Conflict

After modifying the local AST by changing the source code in the local workspace the AST has to be merged during commit with the current AST from the server which likely has also been changed by other users. A direct merge conflict occurs when the merge cannot be executed without deciding which of the both changes (the local or the change from the server) shall be used. A change is an attribute change, the deletion or creation of a relation or the deletion of an AST node. The creation of an AST node is not relevant because no one else can change this new node.

For example, two developers change the attribute "name" of a *Method Declaration* at the same time then a merge is not possible because the merger cannot decide which attribute value is the correct one.

Rule 1:

Attribute change: The AST node which owns the attribute needs an exclusive lock.

If the developer deletes an AST node then the complete tree with all components are deleted too (e.g. deleting a *Method Declaration* deletes also the content of the method). Deleting and changing an AST node at the same time cause again a conflict.

Rule 2:

Deletion of an AST node: The AST node and all component AST nodes below need an exclusive lock.

Next, we want to examine the creation of an AST node for example a Java class. A class is created below a package. To make sure that the package will not be deleted by another developer we need a lock for the package. An exclusive lock would prevent other developers to create a new class in parallel which would be not a problem. Therefore we introduce the concept of a shared lock and the package only has to be shared locked.

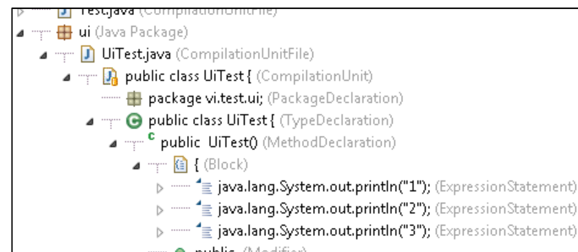

Figure 1: Example of an Abstract Syntax Tree.

In the AST example (see Figure 1) there are three *Expression Statements* below a *Block*. The curly braces indicate a *Block* in Java which can contain one more statements. The order of these statements is relevant. In this example the output on the console should be "1", "2" and "3". If a developer adds a new *Expression Statement* to print "4" after "3" and uses only a shared lock on the *Block*, another developer could add at the same time an *Expression Statement* to print "finished" after "3". In this case we have a merge conflict because the merger cannot decide whether "4" comes after "3" or "finished". If the order is relevant of the AST nodes we need an exclusive lock. The order of the classes in the package is not relevant. Also a changed order of the methods in a class cannot provoke a syntax error and will not change the behaviour of the software.

Of course someone might argue that the developer wants to see the methods in a certain order in the class. However in which order can be different from one use case to another. Nowadays the source code is presented in a very rigid view which is dictated by the source code text. The source code could be presented in different views for example all methods of different classes which are relevant for a certain aspect of the software (Chu-Carroll, 2000). The AST node *Method Declaration* could get some additional attributes (e.g. categories) which define the order of the methods for different use cases.

Rule 3:

Creation or deletion of a composite relationship with the cardinality 0..n: If the order of the

component nodes is relevant, the composite node needs an exclusive lock otherwise a shared lock.

Order relevant relations are for example statements in a method or parameters in a method declaration.

Next, we want to derive a class from a base class. Therefore a *Type Reference* is created below the *Type Declaration* (the Java class). Of course the class can be derived from only one base class. If one developer derives the class from the base class B1 and another developer in parallel from B2 then we have a merge conflict. The merger cannot decide which base class is the correct one. In this use case we need an exclusive lock for the class.

Rule 4:

Creation or deletion of a composite relationship with the cardinality 0..1: The composite node needs an exclusive lock.

For the interfaces the third rule is applicable and because the order of the interfaces is not relevant only a shared lock is needed for the class when adding or removing an interface.

Next, we want to change the content of a method. We have already established that the statements in the method are order relevant. For inserting new statements, deleting statements or changing the order of the statements we need an exclusive lock for the composite node *Block* (see Figure 2).

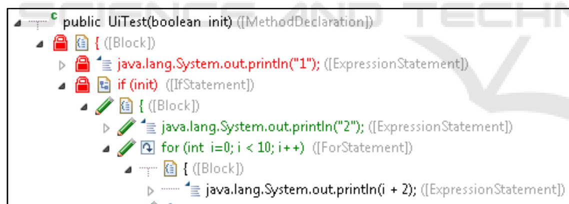

Figure 2: AST nodes of a *Method Declaration*.

Changing an existing statement could be done without an exclusive lock of the *Block*. However to simplify the logic we also acquire an exclusive lock for the *Block* because when the developer changes code the probability is high that new statements are created or existing statements are deleted. If the *Block* is exclusive locked then we can lock all statements below the *Block* because nobody else can change a statement below the *Block*. If there is a statement with a new *Block* (for example a *For Statement* or an *If Statement*) then it is not necessary to lock this *Block* too. With a new *Block* a new ordered list of statements starts and therefore locks of other developers are possible. In Figure 2 we can see that one developer has locked the statements below the *Method Declaration* (in red colour) and another developer the statements below the *If Statement* (in green colour). The *Block* below the *For Statement* can be locked by some else.

Rule 5:

Inserting, deleting or changing statements inside a Block: Starting with the composite node Block of the statement all component nodes and the Block node itself need an exclusive lock until the next Block nodes.

4.2 Indirect Conflicts

An indirect conflict occurs when after the merge of the ASTs a syntax error has been created. This is the case when the supplier node in an association is changed and the client node has not been adapted. If the association already exists and the developer changes the supplier node then the same developer has to correct all client nodes before a commit of the AST is possible because a commit of an AST with syntax errors is not possible. For the adaption the developer needs exclusive locks for the client nodes. There is no additional rule necessary. However in the case of the creation of a new association we have to make sure that the supplier node will not be changed or deleted. This can be achieved via a shared lock. For example when several developers want to call the same method every developer receives a shared lock for the method. With the shared lock there is no modification of the method possible.

Rule 6:

Creation of a new association: the supplier node needs a shared lock.

The following special use cases have to be considered:

4.2.1 Method Override

With the annotation "@Override" in front of a *Method Declaration* it is well defined that the *Method Declaration* overrides another *Method Declaration* from a base class or interface. If the *Method Declaration* in the base class or interface changes (e.g. the parameters) the *Method Declaration* from the derived class has to be adapted. Therefore we need an additional association in the meta-model to express this relationship. With this additional association and rule 6 we can prevent indirect merge conflicts.

4.2.2 Default Constructor

In a Java class without any constructor Java provides a default constructor with no parameters which can be used to instance the class. However the AST node *Class Instance Creation* needs always a constructor for the association "constructor invocation". Therefore during the creation of a new class the according AST nodes are automatically created for the default constructor which then can be used in the association. Indirect merge conflicts are prevented with the already defined locking rules.

4.2.3 Return Statement

If the return type of a *Method Declaration* is changed then all *Return Statements* have to be adapted, too. For example if the return type is changed from void to integer then all return statement have to be changed from "return" to "return <*integer*>". There is a kind of association between the two AST nodes. This association has not been added in the meta-model but we need an additional rule.

Rule 7:

Addition of a new Return Statement: the composite Method Declaration needs a shared lock.

4.3 Name Conflicts

There is an additional potential conflict which we have not yet considered. In Java a package, a class or a method provides a namespace. Inside the namespace names have to be unique for example it is not possible to have two methods with the same name in the same class. Because we use only a shared lock for the *Type Declaration* it is possible that two developers create two *Method Declarations* with the same name. The following two solutions are possible: Firstly, we use an exclusive lock for the *Type Declaration*. Then only one developer can create a new method inside a class. Then of course parallel editing would be much more restricted. Secondly, we introduce a new feature which allows us to reserve or lock a name inside a namespace. These locks could be stored along with the standard locks in the server.

Name conflicts occur less often because the probability that two developers choose exactly the same name in the same name space at the same time is not that high. Also the conflict can be resolved very easily by just renaming the artefact. Therefore we allow this kind of merge conflicts and we did not implement one of the suggested solutions. However we have to ensure that the conflicts are detected and no syntax errors are stored in the server. Therefore the *Compilation Unit* has to be exclusive locked immediately before the commit which will cause an update of the *Compilation Unit* content and reveal any name conflicts.

5 INTEGRATION INTO THE EDITOR

The developer should not bother about locking the Abstract Syntax Tree (AST) nodes according to the lock rules. Therefore we need a good integration of the locking functionality into the Java AST editor.

The Java AST editor knows which AST nodes are located at a certain position in the editor. When the developer starts editing the source code text the Java AST editor determines all AST nodes to be locked by using the lock rules and requests the locks from the server. This is done immediately with the first key stroke because the developer might not retrieve the locks because in the meantime someone else has already locked the AST nodes. In this case the developer should not be able to continue changing the source code text.

In the editor the areas of text which cannot be changed because of foreign locks are displayed with a grey background. In these lines no key strokes are accepted from the editor. The developer can also open a popup window on top of these lines which shows the following information about the lock: lock type, owner of the lock and since when the AST node is locked (see Figure 3).

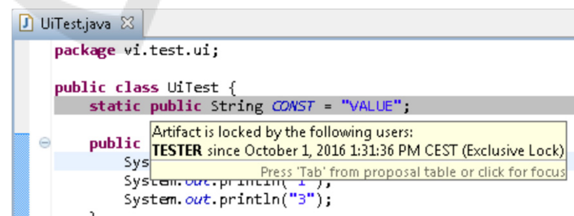

Figure 3: Java AST Editor with lock information.

6 RELATED WORK

The different concept to solve or to improve the handling of collaboration conflicts can be classified into intrusive and non-intrusive strategies (Levin, 2015). Intrusive strategies automatically update the private copy of source code in the local workspace with the source code of other developers. Non-

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

intrusive strategies only inform the developer about the current changes other developers to indicate a potential merge conflict (awareness enhancers) or supports the developer during merging.

6.1 Non-intrusive Strategies

One approach is to analyze changes from different branches within a sequence of changes and to support the integrators work by not just using the text but also the AST (Gómez, 2014).

Other concepts belonging to this strategy propagate information about current changes in the local workspace between team members. There are a number of tools available: Syde (Hattori, 2010), CollabVS (Dewan, 2008), Palantír (Sarma, 2003) and others. Several tools only consider direct merge conflicts.

Palantír is a workspace awareness tool which provides developers insight into other workspaces and was original developed to detect direct merge conflicts. Palantír has then been extended to detect also indirect merge conflicts (Sarma, 2007). Therefore a six-step process has been introduced: collecting, distributing, analysing, informing, filtering, visualizing. In the analysing step the changes in the local workspace and in the remote workspaces are brought together to find any potential indirect conflicts. This is done by examining the dependencies of the remotely changed artefact, both forwards and backwards and then Palantír verifies if any local changed artefacts are involved.

These concepts collect relevant information and then present the information to the software developer. The developer can use or ignore this information. In contrast our approach proactively prevents any conflict before it actually occurs and it is not up to the user to react on the conflict. The underlying assumption is that the developer would rather wait to change the source code than to undergo the effort of a manual merge.

6.2 Intrusive Strategies

Concepts belonging to this strategy propagate the source code between team members and automatically synchronize the source code of the developer with the changes of other developers. There are a small number of projects which implement this strategy for example CloudStudio (Nordio, 2011), Collabode (Goldman, 2011) and CSI (Levin, 2013). This project implements the Synchronized Software Development (SSD) approach.

The CSI solution is an Eclipse plugin. It differs from the other two by supporting a pessimistic locking concept. When a developer is editing a semantic element (e.g. a method) other developers cannot change the same sematic element. While blocked, of course other elements can be changed. This blocking functionality also considers indirect merge conflicts. So, it is not possible that one developer changes a declaration (e.g. of a method) and another developer changes in parallel the source code which depends on the declaration (e.g. the method call). The smallest unit which can be locked is a method and not an Abstract Syntax Tree (AST) node. Therefore it is not possible that two developers work on the same method. When the source code is in a state where no compilation errors are present then the code is propagated automatically to all team members. The developers work on the same unified code version. Several but not all possible scenarios are supported: creating, deleting, renaming or changing a method or a member field.

In contrast our approach is not an intrusive strategy because the software developer decides when his or her changes are published by committing the source code. It is possible to finish a complete feature or parts of a feature before releasing the source code to other developers. Also CSI does not support any shared lock concepts which allows more parallel changes.

7 CONCLUSION & FUTURE WORK

By working with a model (Abstract Syntax Tree - AST) instead of source code text it is possible to introduce a fine grained pessimistic locking concept on artefact level. Direct and indirect merge conflicts which causes syntax errors are completely eliminated. The developer can commit his or her changed source code at any time without any merging effort. Also syntax errors in the repository are no longer possible. A continuous integration (CI) run will never be blocked because of syntax errors in the repository. The fine grained pessimistic locking allows parallel changes in the same class or method by different software developers. The integration of the locking functionality into the Java AST editor enables the developer to automatically lock all necessary AST artefacts without paying attention to the details.

Our future work includes the usage of Morpheus with the fine grained pessimistic locking concept in a large software development project with several million lines of code. The performance and memory issues which arise from processing millions of AST artefacts have to be analysed and solved.

REFERENCES

Perry, D. E., Siy, H. P., & Votta, L. G., 2001. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM), 10(3),* 308-337.

Estublier, J., & Garcia, S., 2005. Process model and awareness in SCM. In *Proceedings of the 12th international workshop on Software configuration management* (pp. 59-74). ACM.

Conradi, R., & Westfechtel, B., 1998. Version models for software configuration management. *ACM Computing Surveys (CSUR), 30(2)*, 232-282.

Grinter, R. E., 1995. Using a configuration management tool to coordinate software development. In *Proceedings of conference on Organizational computing systems* (pp. 168-177). ACM.

Sarma, A., Noroozi, Z., & Van Der Hoek, A., 2003. Palantír: raising awareness among configuration management workspaces. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (pp. 444-454). IEEE.

Levin, S., 2013. *Synchronized software development* (Doctoral dissertation, Tel-Aviv University).

Sarma, A., Bortis, G., & Van Der Hoek, A., 2007. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (pp. 94-103). ACM.

Dewan, P., 2008. Dimensions of tools for detecting software conflicts. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering* (pp. 21-25). ACM.

Levin, S., & Yehudai, A. (2015). Alleviating Merge Conflicts with Fine-grained Visual Awareness. *arXiv preprint arXiv:1508.01872*.

Broschy, P., 2009. Improving conflict resolution in model versioning systems. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on* (pp. 355-358). IEEE.

Brun, Y., Holmes, R., Ernst, M. D., & Notkin, D., 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (pp. 168-178). ACM.

Eyl, M., Reichmann, C., & Müller-Glaser, K., 2016. Fast Feedback from Automated Tests Executed with the Product Build. In *International Conference on Software Quality* (pp. 199-210). Springer International Publishing.

Eclipse, 2016. „Eclipse Neon“. Eclipse.org. Retrieved 2016-07-30 from http://eclipse.org.

OMG, 2016. „OMG's MetaObject Facility (MOF) Home Page“. Omg.org. Retrieved 2016-07-30 from http://www.omg.org/mof/.

JetBrains, 2016: „MPS overview“. *JetBrains*. Retrieved 2016-07-30 from https://www.jetbrains.com/mps

Hattori, L., & Lanza, M., 2010. Syde: a tool for collaborative software development. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* (pp. 235-238). ACM.

Nordio, M., Estler, H., Furia, C. A., & Meyer, B., 2011. Collaborative software development on the web. *arXiv preprint arXiv:1105.0768*.

Goldman, M., Little, G., & Miller, R. C., 2011. Collabode: collaborative coding in the browser. In *Proceedings of the 4th international workshop on Cooperative and human aspects of software engineering* (pp. 65-68). ACM.

Chu-Carroll, M., & Sprenkle, S., 2000. Software configuration management as a mechanism for multidimensional separation of concerns.

Vector, 2016: „PREEvision – Development Tool for model-based E/E Engineering“. Vector.com. Retrieved 2016-07-30 from https://vector.com/vi_preevision_en.html.

Zhang, R., & Krishnan, A., 2011. Using delta model for collaborative work of industrial large-scaled E/E architecture models. In *Model Driven Engineering Languages and Systems (*pp. 714-728). Springer Berlin Heidelberg.

Gómez, V. U., Ducasse, S., & Kellens, A., 2014. Supporting streams of changes during branch integration. In *Science of Computer Programming* (pp. 84-106), 96.