Masterarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Improving the reliability of Saros using Root Cause Analysis

Sebastian Starroske
sebastian.starroske@fu-berlin.de

Berlin, 14. Januar 2013

Gutachter:	Prof. Dr. Lutz Prechelt
Zweitgutachterin:	Prof. Dr. Claudia Müller-Birn
Betreuer:	Franz Zieris

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Sebastian Starroske

Berlin, 14.01.2013

Abstract

In this thesis a Root Cause Analysis will be performed with the goal to improve the reliability of a tool called Saros. Saros is an Eclipse plug-in that can be used for distributed pair programming.

At the beginning of the thesis the concepts of Root Cause Analysis and the way how they are applied at the project Saros will be described. Some basic facts about Saros, its features as well as some basic architecture properties are explained afterwards. The main section of this thesis covers the data collection and defect correction. Apart from resolving some of the identified defects an overview is created that shows which failures are caused by which defects. The identification and resolving of root causes was not completed in this thesis.

Contents

List of Figures

Listings

1 Introduction

One field of research of the working group Software Engineering (AG SE) of the Freie Universität Berlin is pair programming. There are basically two types of pair programming: the traditional one where two programmers work together in a pair in front of the same workstation and the distributed pair programming (DPP) where the two programmers see the same part of code on individual screens [SS00]. In order to study DPP the AG SE started to develop a tool which enables programmers to perform DPP. The tool is an Eclipse[1] plug-in, called Saros.

Saros has been developed for about 7 years up to day. The main part of the development was done in the scope of so called X-theses[2]. At the beginning only two developers could work together on one Eclipse project. Later on many features were added. It is now possible to work on multiple projects with more then two users which can simultaneously edit the shared resources.

In order too analyze DPP in practice Saros needs to be reliable enough, so that it can be used by companies and other institutions. Therefor improving the reliability is the main goal of this thesis. There are different approaches to achieve this goal, of which the Root Cause Analysis was chosen.

In the scope of Saros the most important aspect of reliability is consistency. It is the main precondition for DPP. Consistency means in the case of Saros that the files on each participants side have the same state.[3]

[1] Eclipse is an integrated development environment(IDE) which can be used for many different programming languages. Website: http://www.eclipse.org

[2] master theses, bachelor theses or diploma theses

[3] The same state means that after some changes are applied by both developers the contents of the files on both sides should be equal.

2 Goals

The main goal of this thesis is to improve the reliability of Saros. This means that basic operations like editing, moving and deleting files do not cause inconsistencies. If any inconsistencies should occur, Saros needs to be able to detect and correct them. In order to accomplish this main goal several sub goals are defined.

To improve the reliability of Saros it is important to analyze all existing problems which can cause inconsistencies. Therefor naming and describing all current inconsistency types in Saros is one sub goal.

A Root Cause Analysis (RCA) should be performed for the inconsistency types with the highest risk value. The outcome of the RCA should be suggestions how some inconsistencies can be prevented in the future, as well as a recommendation how future RCAs should be performed in Saros. The solutions for the found root causes need to be implemented as far as it is possible. The root causes which can not be fixed have to be well documented, so that they can be fixed by other Saros Developers in the future. A nice-to-have would be an analysis how the solved root causes reduce other non-inconsistency related problems.

There are several other existing problems in Saros, but they are not scope of this thesis. This thesis focuses on inconsistency related issues, because all Saros features need a consistent state of the session to function accordingly. Besides some inconsistencies can not be resolved so that a restart of the session is necessary. This can be really annoying for the users and might be a reason why they will not continue working with Saros.

3 Methodology and Approach

In this section I want to introduce Root Cause Analysis (RCA) and explain why and how I will use it to improve the reliability of Saros.

3.1 Root Cause Analysis

Root Cause Analysis is a process which can be used to analyze why certain events, which have a negative impact on a product or process, occur [JJR04]. Referring to Software such an event would be a failure. The goal of RCA is to use the course of events in order too prevent accidents or failures to happen in the future. The idea is that for each failure a so called Root Cause (RC) can be identified.

Rooney and Heuvel define a root cause as a specific underlying cause, which can be reasonably identified. With reasonably they mean that the benefit of identifying the root cause is comparable too the effort it costs. That also means that at a certain depth it makes no sense to try to analyze further. Another attribute of root causes is that they can be controlled. For example: it would make no sense to blame the education system for failures which occur in a Software because the project Team can not change the education system. Removing a root cause can not only resolve the resulting failure but also prevent future ones. That way RCA can help to improve the reliability as well as the stability of a software.

A RCA consists of the following four steps [JJR04]:

1. Data collection
2. Causal factor charting
3. Root cause identification
4. Recommendation generation and implementation

Data Collection
In this step all needed data is collected. That includes detailed information about the observed failures and how they can be reproduced. Identifying the direct causes of those failures is also part of this step.

Causal factor charting
In this step the collected information is structured and sequence diagrams are used to model a cause effect chain. The diagrams can be used to identify major contributors to the failures, the so called causal factors.

Root cause identification
In this step the root causes are identified. This is done by analyzing how and why the causal factors occurred.

Recommendation generation and implementation
In the last step of the RCA recommendations on how to fix the identified root causes are

made. It is also possible to already implement the recommendations and fix the root causes right away.

3.2 Approach

After the basics of RCA have been described in the previous section the focus of this section is why and how I will use RCA to improve the reliability of Saros.

When I started to work on this thesis there were more than 20 open bug entries with the category "consistency" were registered in the SourceForge bug tracker of Saros[4]. Instead of just dealing with the symptoms of the failures described in the bug entries, it would be better to fix the defects that caused them. An even better approach would be to identify the root causes which led to the defects, because this could prevent similar defects in the future and would therefor increase the stability of Saros.

In order to perform a RCA I need to find a way to apply the basic concepts described by Rooney and Heuvel on the Saros project. The problem is that the concept of RCA is very generic and can be used for software, in healthcare or manufacturing. I have not found useful examples of the application of RCA in the Software Development which I could use to deduce a concrete process.

In the first step - the data collection - I will collect all available information about the known failures from the bug tracker and collect them in a table. This table contains the following information for each bug entry:

- Can the failure/s described in the bug entry be reproduced?
- Is the entry still open or already closed?
- When was the bug entry created?
- Which features and module are related to the described failures?

After that I will divide the bug entries on the base of the collected data into clusters of bug entries with similar properties. I need to select one of these clusters to start with and try to find the defects that led to the described failures.

In the second step - the causal factor charting - I will create a topographical map which shows the observed failures on the top and the main defects on the bottom. I will show in this chart which defects led to which failures in section 5.3.

The next step is to identify the root causes on the product level by analyzing when the major defects were introduced and why they occurred. If possible I will try to resolve these root causes and extend my RCA to the process level. To do so I need to analyze what type of error caused the defects on the project level and how they are connected with the used development process.

[4] the bug tracker can be found here: http://sourceforge.net/tracker/?atid=843359&group_id=167540&func=browse

4 Saros

In this chapter I will describe some features, components and architectural details of Saros, which will be needed to understand section 5. This includes information about Activities, the Consistency Watchdog, the Jupiter implementation as well as the features Partial Sharing, Needs-Based Synchronization and Multiuser session.

4.1 Activities

During a session the main communication between the session participants is done via Activities. Activities are messages which are created when events like text edits or the deletion, creation or renaming of files occur. Every Activity class implements the `IActivity` interface and has a source attribute which represents the user who caused the initial creation of the Activity instance.

Some Activities are resource-related and therefore implement the `IResourceActivity` interface. Instances of this type contain a path object which contains a project and a project relative path. Another type of Activities implement the `ITargetedActivity` interface. These are Activities, which are to be sent to a certain group of users.

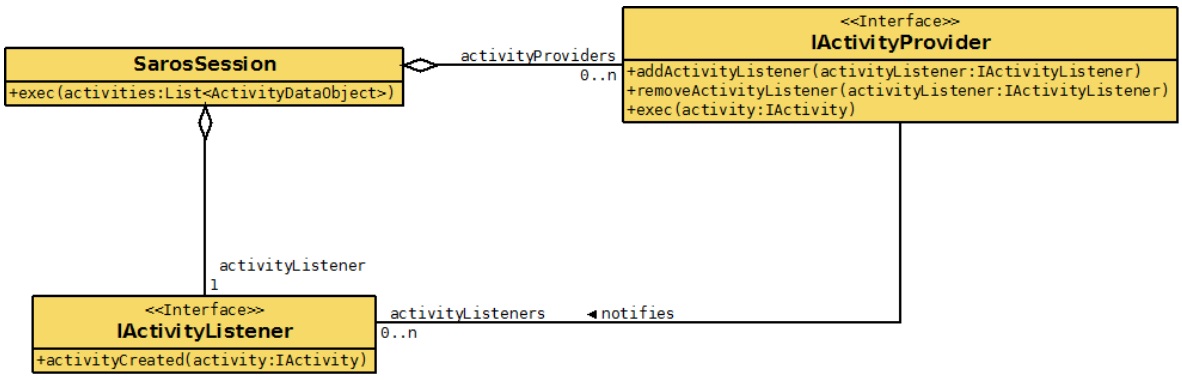

Figure 1: Activity provider and listener

Figure 1 shows a class diagramm containing important classes which participate in sending, receiving and executing Activities. The `SarosSession` is responsible for sending and receiving. It can be seen in the class diagram that the SarosSession has a list of registered Activity Providers which can execute the received Activities. A provider is responsible for certain Activity types and should ignore all other types. Besides the execution of Activities an Activity provider can also create new Activities. When a new Activity is created the provider notifies the internal Activity Listener of the `SarosSession`, which will initiate the sending of the new Activity.

Before an Activity is sent via the network it is transformed in a serializable Object. This Object works like an envelope. It contains the data of the Activity in way that it can be

transmitted while hiding the information from the network layer. After such an envelope is received by a session participant it will be transformed back into an Activity.

4.2 Consistency Watchdog

The consistency watchdog is responsible for detecting and correcting any inconsistencies which might occur during a session. The session host owns the data sovereignty, that means any shared files on the client side which differ from the corresponding files on the host side are considered inconsistent. There are different ways how these inconsistencies could occur and not all are known and can be reproduced. Inconsistencies can occur if Activities were not submitted correctly or if the execution of the Activities took too long. Another example would be if files are altered outside of Eclipse (e.g. a webserver which writes log files).
The consistency watchdog consists of two components, one is on the side of the session host and the other one is running on the participant's side.

The `ConsistencyWatchdogServer` is a job which is scheduled every ten seconds on the host side. The job calculates the checksums of all files which are currently opened by any session participant and sends them via the network using `ChecksumActivities`. The `ConsistencyWatchdogClient` is an ActivityProvider which can receive `ChecksumActivities`. The client watchdog compares the checksums received from the host with the calculated checksums of the local files. If the checksums differ an inconsistency is detected and the local user gets notified via a balloon notification. The inconsistency button is enabled as well.

If the user clicks on the inconsistency button the `ConsistencyWatchdogClient` starts the recovery process. The client sends a `ChecksumErrorActivity` to the host, signaling that he has detected inconsistencies. The `ChecksumErrorActivity` contains a recovery ID which is used to identify the current recovery process as well as a list with the inconsistent files. When the host receives a `ChecksumErrorActivity` he blocks all participants and starts sending the requested files to the inconsistent participants using `FileActivities`. After all files are submitted the host sends a `ChecksumErrorActivity` back to the inconsistent user to inform him that all requested files have been sent and also unblocks all participants.

4.3 Jupiter Algorithm

The Jupiter algorithm is used to enable concurrent editing. In a Saros Session each participant has his own copy of the shared files. All changes a user applies need to be synchronized to all other participants. This is very complicated if the participants are typing at the same time.

In order to solve this problem the algorithm is applied for each shared resource separately. For each resource there exists one Jupiter server and one Jupiter client for each participants. That way the Jupiter algorithm can keep track of the state of each user by introducing a 2-dimensional state space. Each document has a Jupiter timestamp which consists of two values: the number of the current remote operations (server document) and the number of local operations (document on the client side). With this information the server can compute individual operations which need to be applied by each user in order to get back to a

consistent state. The operations are sent via JupiterActivities.

4.4 Partial Sharing

At the beginning of the development of Saros it was only possible to share exactly one project. In Q1 2011 the sharing was expanded to multiple projects and it was also possible to add projects during a running session. The next step was the implementation of the feature called Partial Sharing. This feature enables the sharing of a set of files instead of a whole project. The motivation was to avoid long waiting times which can occur when large projects are shared. Sometimes it is enough to share only files belonging to a certain module instead of sharing the complete project.

4.5 Needs-Based Synchronization

The Needs-Based Synchronization is an optional feature, which adds functionality for the user's convenience in the context of partially shared projects. It consists of two elements: Pre-Sharing and Quick-Adding.

Quick-Adding enables a user to add files of a partial shared project, which have not been shared yet, to a running Session. The first time a user opens a non-shared file he is asked if he wants to add the file to the running session. He also can choose to automatically add files in the future.

The other part of the Needs-Based feature is Pre-Sharing. If the inviter opens a file during the project synchronization, the file is shared first, so that the participants can start working on it even before the synchronization is finished.

5 Invitation Process

In this chapter I want to describe and analyze defects which are connected with the invitation process. I will start describing the original invitation process, as well as the observed failures. I will continue with a description of my solution process and with the identification of root causes.

5.1 Original Invitation Process

The invitation process as it is today is similar to the process in very early Saros version in 2006. It still consists of the same steps which are executed in the same order. Djemili divides the invitation process into six phases [Dje06, page 83 ff.]. In the first phase the inviter chooses a contact and sends an invitation. The invitee can accept the invitation in the second phase. If he does, the inviter sends a list containing all files he wants to share. In the fourth phase the Saros processes the list of files on the invitee side and determines which files are needed by the invitee. The resulting list is returned to the inviter. The files are transmitted in the fifth phase and are received and processed by the invitee in the sixth phase.

The current invitation process as of January 2013 is shown in figure 2. The main difference from the old invitation process is that it is divided into two sub-processes: the Session and the project invitation. Back in 2006 only two users were working together on only one shared project in a Saros Session. Now it is possible to work with multiple users on multiple projects as well as to add additional projects during the running Session. In order to achieve this, Dohnert split the invitation process into the Session and project invitation [Doh11, page 10 ff.].

In the case that user Alice invites user Bob to work on project A the complete process displayed in figure 2 is followed through. It starts with the Session invitation which covers the first two phases that Djemli has described, but now Bob is already visible for all other participants in the Roster View and he is also member of the Session's chat room. The project invitation consists of the phases three to six. First the so called ProjectExchangeInfos, which consist of the names of the shared projects and the paths of the shared files, are sent to the invitee who has now different choices how to proceed with them. For each project he can choose to create a new project for the shared files or to use an existing project in his workspace in order to reduce the amount of files which need to be transmitted via the network. In the latter case all files which differ from the ones on the inviter side (or don't exist) will be overwritten during the synchronization. While the project archive is created on the host side all Session participants are blocked which means that they can not make changes to any shared resources. If Alice wants to add project B to the current Session with Bob, there is no need anymore for the complete invitation process. Only a project invitation takes place.

5.2 Observed Failures

In section 3.2 I described how I want to approach the RCA. The first step is the collection of data which includes the description of the occuring failures and how they can be

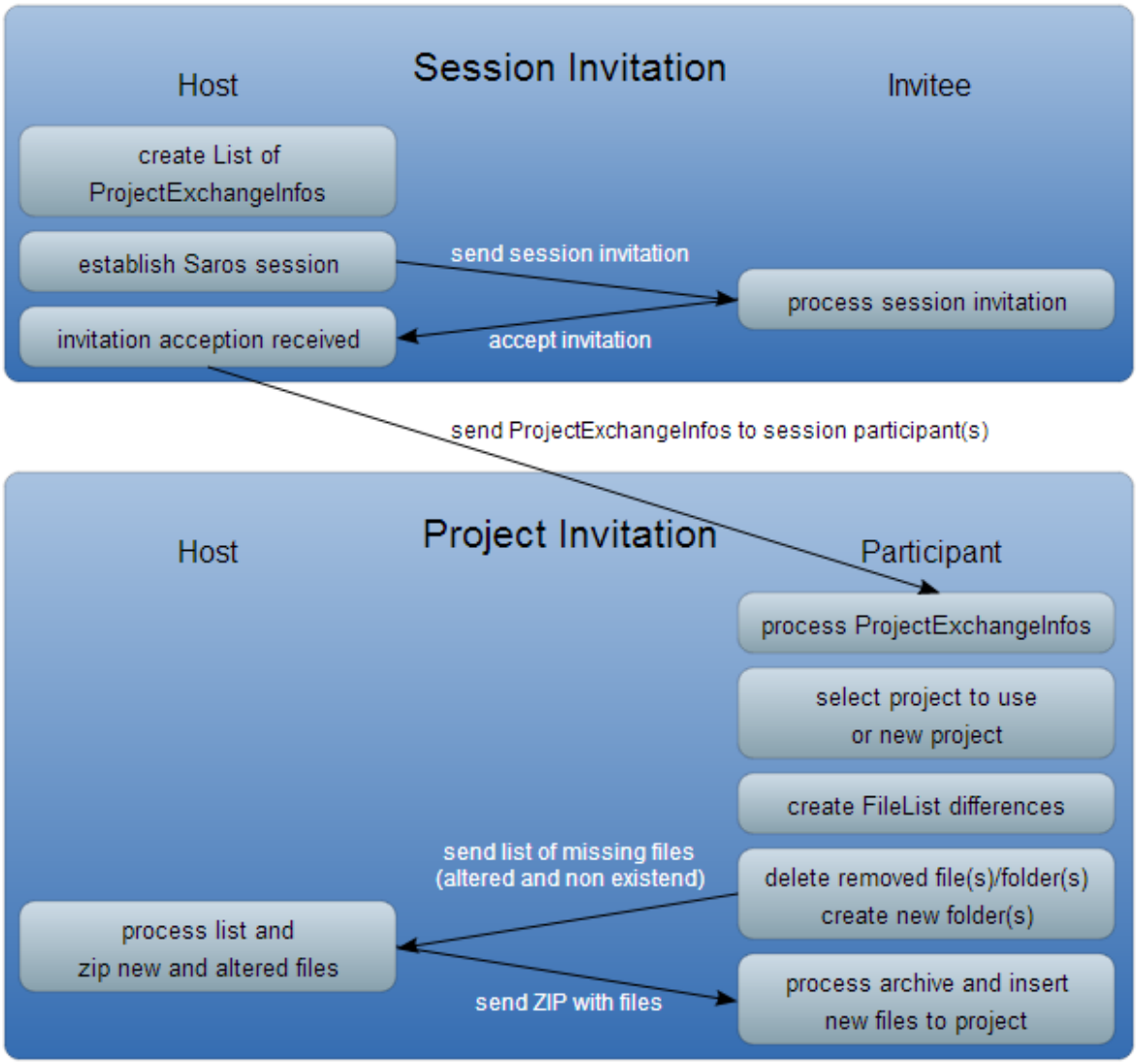

Figure 2: Invitation Process [Sar11]

reproduced. This is done in this section. I want to describe the failures I observed concerning the invitation process and I will refer to them as F1 ... Fn in the remainder of this chapter.

I used the entries of the SourceForge bug tracker with the IDs 3458952[5] and 3512804[6] as the starting point for my analysis. I will refer to these entries in the following paragraphs with **scenario A** (3458952) and **scenario B** (3512804) Both entries describe a similar course of events:

1. The users Alice(host) and Bob are currently working together on a project using Saros.
2. In case of scenario A the host(Alice); in case of scenario B the non-host (Bob) invites Carl.
3. During the invitation process the non-inviting participant continues to work on the project (insert and remove text and files).
4. After the invitation is finished the touched files are inconsistent on Carls's side and the inconsistencies won't be detected or if they are, Saros won't be able to fix them.

There are two differences between those scenarios: The first is the role of the inviter (host or non-host) and the second is that in scenario A the needs-based feature is activated. Activating the needs-based feature results in the error message shown in figure 3, which is displayed on the host side. This is really confusing for the user, because she has not edited a non-shared file. If she confirms with "Yes", all changes Bob has made are overwritten and Bob becomes inconsistent. The misbehavior of the needs-based feature during the invitation is the first failure **F1**.

If the needs-based feature is disabled (like in scenario B) or if Alice clicked "No" in the needs-based dialog there will still occur inconsistencies on Carl's side (the person that gets invited). This is the second failure **F2**. In any case those occurring inconsistencies can not be detected by the watchdog (failure **F3**). Besides, Carl can neither send nor receive any text edits, which indicates, that the Jupiter algorithm does not work anymore (failure **F4**)

To sum up this section: if a Session participant continuous to work (e.g. edit files) on a project while another user is invited to the session, four different failures can occur:

- **F1:** The needs-based dialog is displayed when it is not intended to and it provides wrong and confusing information to the user.
- **F2:** After the invitation process the files are inconsistent on Carl's side.
- **F3:** The inconsistencies can not be detected and resolved by the watchdog.
- **F4:** The Jupiter algorithm does not work for Carl anymore and prevents him from sending and receiving text edits.

[5] http://sourceforge.net/tracker/?func=detail&aid=3458952&group_id=167540&atid=843359
[6] http://sourceforge.net/tracker/?func=detail&aid=3512804&group_id=167540&atid=843359

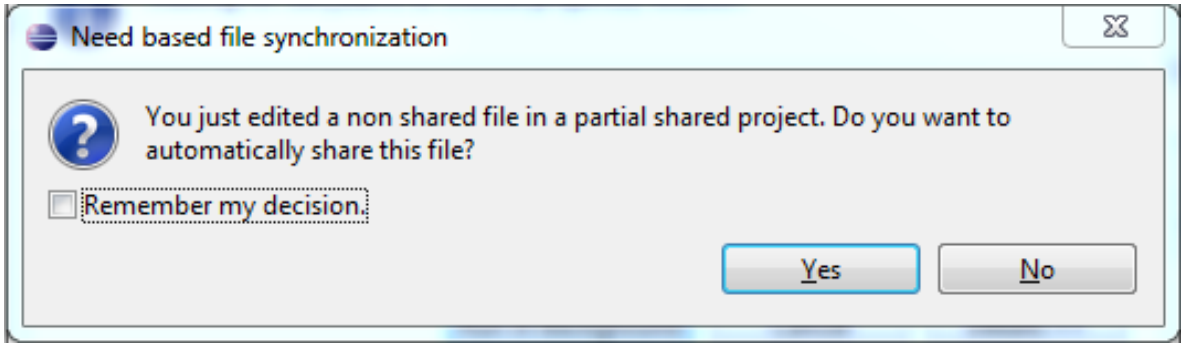

Figure 3: Needs-Based Synchronization error message during an invitation

5.3 Defect Overview

Now that I have described the observable failures I will continue to analyze what defects lead to them and how they are connected to each other. In order to visualize the connections I revealed I will present an overview, like a topographic map of defects and failures. This map will function as my causal factor chart, which should be created in step two of a RCA. At the beginning this map is empty, but as I a describe my progress I will add more information to it. The complete map can be seen in the Appendix: figure 10.

To verify if my implementation avoids the failures F1...Fn I used an existing STF test case[7]. This test case covers the scenarios described in the previous section.

Step 1

As I described in the previous section there are four failures related to the invitation process. My first approach was directed to preventing failure F2 (inconsistencies after invitation). The idea of this approach is not to prevent inconsistencies during the invitation process, but to manually resolve them as soon the invitation is completed. In order to do so the invitee (in the scenarios described above, this would be Carl) collects all paths of the resources that were touched during the synchronization process, by intercepting resource related Activities before they are transformed.[8] These Activities contain the path to their resource which is saved in a Set. At the end of the invitation process the client-side Watchdog is used to resolve any inconsistencies occurring in any of the resources saved in the Set. The Watchdog requests a checksum for each file from the host and compares it with the local checksum. If any checksum differs the Watchdog starts resolving the inconsistency.

This implementation works, but it has some drawbacks. The inconsistencies which can occur during the invitation process are resolved and future sending and receiving of Activities is

[7]STF stands for "Saros Test Framework" and is a framework which is used to write test cases that test Saros in the Eclipse context. This test framework enables the developer to remotely control multiple Eclipse instances and perform operations like creating or sharing a project

[8]That means it is the serializable form of Activities, the so called ActivityDataObjects. Transforming an Activity would basically mean opening an envelope. To keep it simple I will refer to them as Activities, independent if it is an `ActivityDataObject` or an `Activity` implementation

ensured, but to achieve this the Watchdog functionality is abused. As described in section 4.2 the checking for inconsistencies is started from the host side by an automatically repeated task. This process was not intended to be started manually. Another disadvantage is that this implementation creates a dependency between the invitation process and the watchdog, but these modules have nothing in common and should operate independent from each other. Besides the violation of the architecture this solution just fixes the inconsistencies right after they arise instead of preventing them. And finally, it still leaves the question unanswered why the Watchdog can not resolve these inconsistencies using the regular process.

Step 2

Since my first approach was not satisfying I tried searching for what was causing the inconsistencies in the first place. Looking at the log files I noticed that all resource-related Activities which have been sent during the invitation process have caused exceptions on the invitee site. The reason is that these activities can not be transformed and applied if the corresponding resource does not exist yet. This problem should be solved by queuing those activities until the invitation is finished and execute them afterwards. The `SarosSession` receives Activities in the `exec(...)` method. Before an Activity can be executed it needs to be transformed first (opening the envelope), which is done in the `convertAndQueueProjectActivities(...)` method. The first problem with this method is that it is responsible for two different tasks: transforming and queuing. The second problem, which causes the inconsistencies, is that the queuing mechanism is not correct. Only `EditorActivities`[9] are queued, but there are other resource-related Activities which are not queued and therefor will be lost. The solution is to queue all resource-related Activities and execute them after the invitation process is finished and all required projects exist in the workspace of the invitee. I could reduce the amount of Activities that can get lost during the invitation process with this approach, but there were still occurring inconsistencies. The problem is, that the JupiterActivities were not executed in the right order. This can happen if new Activities are created at the same moment as the invitee tries to execute the queued Activities.

Figure 4 shows my knowledge about the problem after step two. So far I have found out, that the failure F2 (inconsistencies after the invitation process) is the result of missing Activities which can get lost during the invitation process. This is caused by defect D2 - the wrong implementation of the Activity queuing. Although I identified the defect, I was not able to fix it in step two. Step three will focus on further improving the Activity Queuing.

Step 3

I order to ensure that all Activities are executed in the correct order, I implemented a queuing mechanism which queues all Activities all the time. As soon as a participant joins a Session the Activities he receives are queued. I decided to use separate queues for different types of Activities. One queue is used for resource-independent Activities which can be executed all the time - since they are not dependent on any resources. For each shared project in a session another queue is created to hold the resource-related Activities.

[9] EditorActivities are created if a user changes, opens or closes an eclipse editor

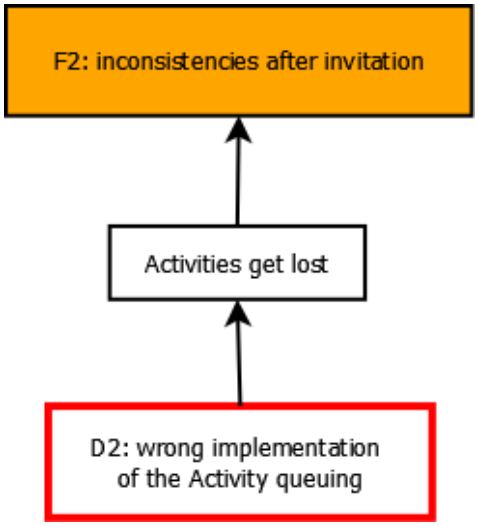

Figure 4: Defect Overview I

`ActivityDispatcherThread`s are used to control the execution of the queued Activities on the SWT Thread. The Dispatcher which is responsible for the resource-independent Activities is started right at the beginning of a Session whereas all other Dispatchers are started after the project invitation is finished and all resources are received. The advantage of this implementation is that no Activity which is sent during the invitation process can get lost since all Activities are queued. In addition the queuing avoids that Activities are executed in a wrong order as it is described in step 2. This implementation improves the queuing and reduces the amount of inconsistencies, but it does not prevent all inconsistencies. Figure 5

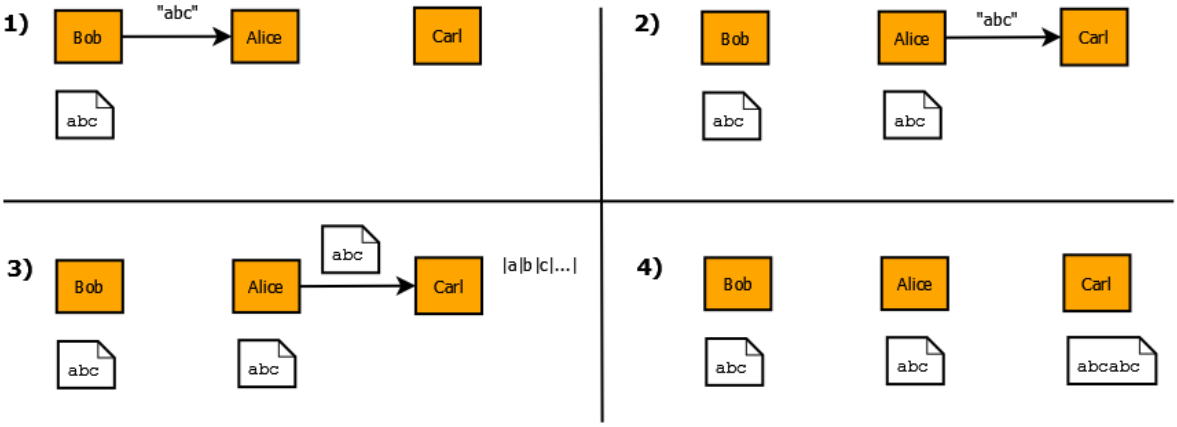

Figure 5: Inconsistency after invitation process

shows a typically scenario which will lead to inconsistencies on Carl's side. If Bob types "abc" during the invitation process (1) a JupiterActivity is sent to the host (Alice) containing the changes by Bob. The changes are applied on Alice's text file and the DocumentServer described in section 4.3 determines that the message "abc" needs to be sent to the other Session participant (Carl) as well (2). Although the invitation process is not yet finished,

Carl is already participating (the session invitation is already completed) and therefor can receive Activities, which will be queued (3). Meanwhile Alice adds the file containing "abc" (since the Activities created by Bob have already been executed) to the project archive and sends it to Carl along with all other file which need to be shared. After Carl has received the project archive he unpacks it and starts executing the queued Activities (4). This results in an inconsistency, because the string "abc" is contained twice on Carl's side. This implementation is still an improvement, because the watchdog can now resolve these inconsistencies.

Step 4

As described in the previous section there are still some problems with the implementation of the Activity queuing. Some Activities are executed twice: once on the host side before the resource is the sent to the invitee and once on the invitee side after the invitation is completed. To avoid this behavior the host needs to be able to decide which Activities have already been applied to the resources before they are packed. He can use this information to send a blacklist to the invitee, containing information which activities should not be applied. The problem is that with the current architecture of Saros it is not possible to identify Activities(for example using IDs). That's why I came up with another solution: The host needs to filter the Activities before he forwards them to the invitee.

In my implementation the host updates a list of resources which have already been added to the project archive while the archive is created. This list is used to decide if an Activity needs to be sent to the invitee. If the path of the resource in the Activity is already in the list, the Activity needs to be sent to the invitee, because the changes caused by the Activity won't be included in the project archive. If the list does not contain the path it means that this resource has not yet been included in the project Archive. Therefor all changes caused by this Activity will be applied before the file is add to the archive, so the Activity needs not to be sent.

This implementation fixed the behavior that Activities were applied twice on the invitee side as I described it in step 3, but now the invitee can not receive any text edits after the invitation which indicates that the Jupiter algorithm does not work anymore (failure F4). The problem is connected to Jupiter timestamps, which are included in the Jupiter Activities. This timestamp is a tuple of the number of remote operations and the number of local operations. When Saros receives a Jupiter Activity local operation count and the remote operation count are checked if they match. If they do, the Activity will be executed and the local operation count will be updated. If the do not match an Exception is thrown and the Activity can not be executed. So if Jupiter Activities are lost or - as in my implementation - not sent on purpose the timestamps get inconsistent and the JupiterAlgorithm does not work anymore. The solution is to send empty Jupiter Activities which update the timestamp but do not contain an operation (like insert or delete).

Figure 6 shows the new insights I gained during my implementations. The fact that Activities can get lost during the invitation process does not only cause inconsistencies (failure F2), but can also cause inconsistent Jupiter timestamp. This leads to failure F2.

Step 5

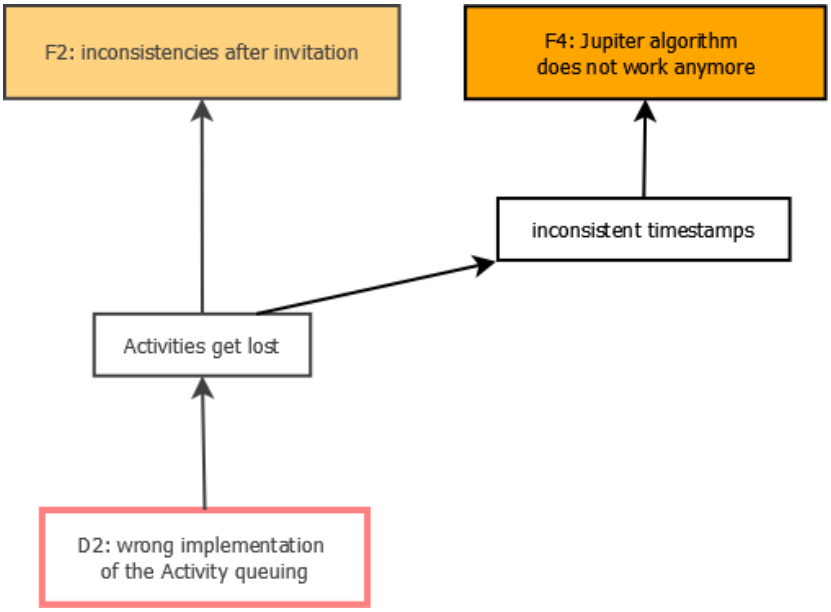

Figure 6: Defect Overview II

When I tested my implementation I still found some inconsistencies. Some inserts were still missing. Analyzing the log files I found out, that the Activites were sent accordingly, but it seemed that the changes to the resources were not saved before they were sent to the invitee. All files opened by the inviter are saved using the static method `EditorAPI.saveProject(...)`[10]. When looking at this method I noticed that it was saving only those resources that were currently opened in an Eclipse editor by the host, but in the STF test case Bob is editing a file that is not opened by Alice. This explains why some characters where missing at the end of the test on Carl's side. I was able to solve this problem by using the `EditorManager`[11] to get all files which are currently opened by any of the Session participants and save them manually.

Figure 7 shows my insights after my implementations done in step 5. As it can be seen in the overview the failure F2 (inconsistencies after the invitation) is caused by two defects. One is the insufficient saving mechanism (D3) and the other one is the wrong implementation of the Activity queuing (D2). This implementation successfully avoids inconsistencies during the invitation process if the host invites another user. The only drawback is, that it does not prevent inconsistencies if the non-host invites somebody to the session, because the non-host can not select which Activities need to be sent. In the next step I will try to further improve the invitation process so that there will be no period where users are blocked.

Step 6

Sóti identified the creation of the project archive which will be sent to the invitee as an

[10] The `EditorAPI` class encapsulates basic interactions with the Eclipse editors like opening and closing editors

[11] The `EditorManager` is responsible for handling user inputs, locking editors and keeping track of the editors which are opened by the local user as well as all remote users

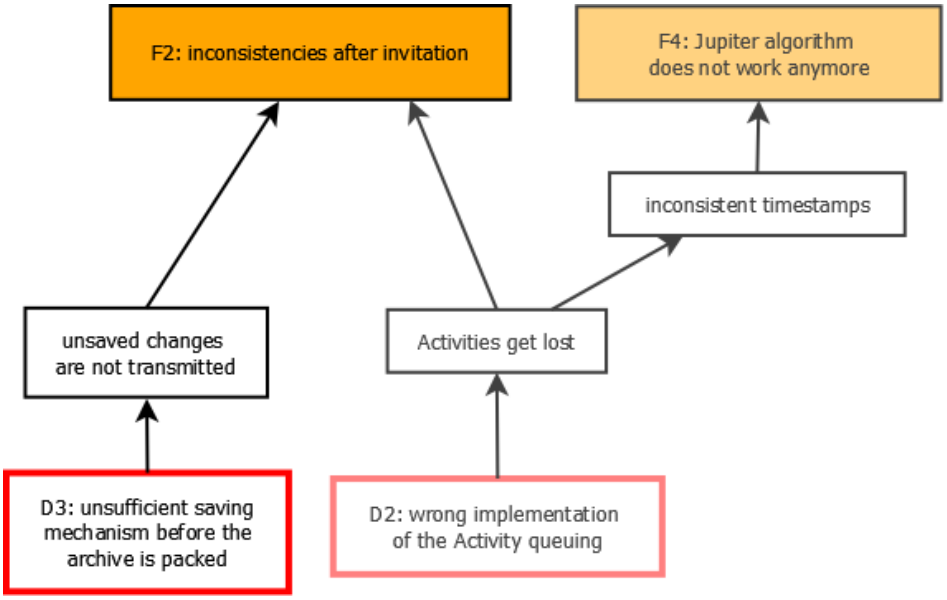

Figure 7: Defect Overview III

critical period where all session participants need to be blocked [Sót09, page 28 ff.]. Since I had already implemented a correct queuing mechanism as well as a correct way to select which Activities should be sent to the invitee and which not, my goal was to get rid of the blocking during the invitation process. In order to achieve this I just needed to remove the blocking part from the project invitation and to save each file before it is added to the project archive.

To verify my implementation I implemented a stress test. It is a STF test similar to the already existing one, but during this test multiple files are modified by Bob during the invitation. This stress test was failing one out of ten times because of inconsistencies and I could not find the reason why. The problem is that the completing of the file list (all resources which have already been added to the archive), and the local execution of the Activities are executed parallel are no elemental operations. This makes it hard to provide a reliable synchronization mechanism.

Step 7

Since I had not successfully managed to avoid blocking the participants during the invitation process I tried to find a much simpler solution. This is possible if the blocking mechanism is used, so I abandoned my queuing implementation and improved the blocking of the users. I found two defects which are related to the blocking mechanism.

One defect is, that the participants are not completely blocked during the invitation process. Although text edits can be blocked, it is not possible to prevent all user inputs. A user could still use shortcuts (e.g. Ctrl + D - delete current line, Ctrl + space - autocomplete) to alter a resource while he is blocked. If he does it must be ensured that no Activities are sent to other users. In order to fix this defect I needed to change the `textAboutToBeChanged(...)` method

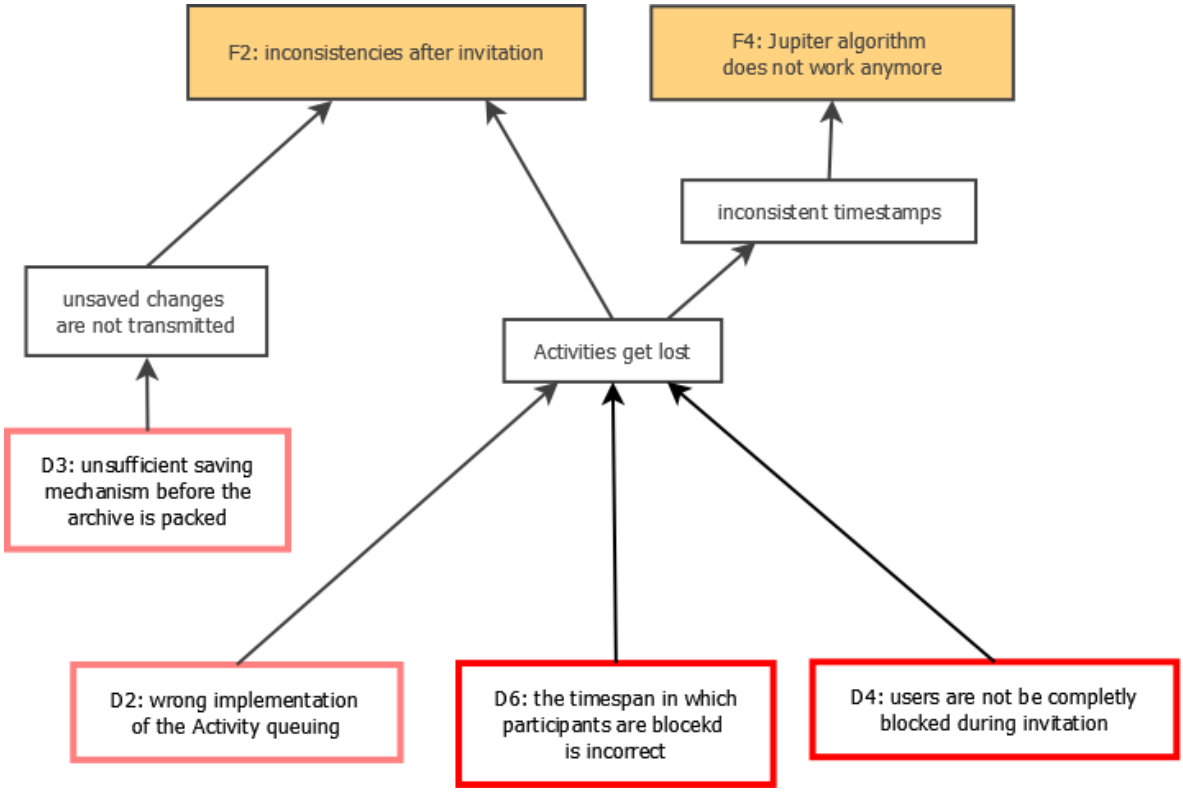

Figure 8: Defect Overview IV

in the `EditorManager` which gets called every time the user changes text in a Eclipse editor. Listing 1 shows an extract of the method. The method creates an Activity containing the information of the changes made by the user and notifies all registered `ActivityListeners` so that the Activity can be sent. I added the if-block which causes the method to return before the Activity is sent if the user is currently blocked (`isLocked`).

```
...
TextEditActivity textEdit = new TextEditActivity(
    sarosSession.getLocalUser(), offset, text, replacedText, path);

if (!hasWriteAccess || isLocked) {
    return;
}
...
fireActivity(textEdit);
```

Listing 1: EditorManager.class

The other defect is that with the current implementation the timespan in which participants are blocked needs to cover the complete project invitation process. After the invitee is done unpacking the project archive he sends a message via the network to the inviter and states that he can unblock all participants. Figure 8 shows an updated version of the defect

overview. As it can be seen the problem that Activities might get lost during the invitation process can be avoided if the blocking mechanism is implemented correctly (D4 and D6).

Occasionally there still might occur inconsistencies, for example if Activities are executed while the session invitation is finished. The participants are not yet blocked and the host will forward this Activity to the invitee. The result are inconsistencies which can not be detected by the watchdog (F3). I found out that this problem is caused if there are any inconsistent Jupiter timestamps. In that case the server-side watchdog will not send checksums for the file with the inconsistent checksums. To fix this I implemented that the Jupiter timestamps for the invitee will be reset after the invitation is completed. As it can be seen in figure 9 if Activities get lost and the Jupiter timestamps of certain resources get inconsistent it also affects the watchdog functionality.

This implementation improves the reliability of Saros, because it prevents inconsistencies if regular text edits are made during the invitation process. If the blocking mechanism is circumvented as described above by using shortcuts inconsistencies can occur on the side of the participant who caused the changes. These inconsistencies will be detected by the watchdog and can be resolved. At the time of deadline of this thesis the patch containing the fix of the defects D5 and D6 are still in the review progress[12].

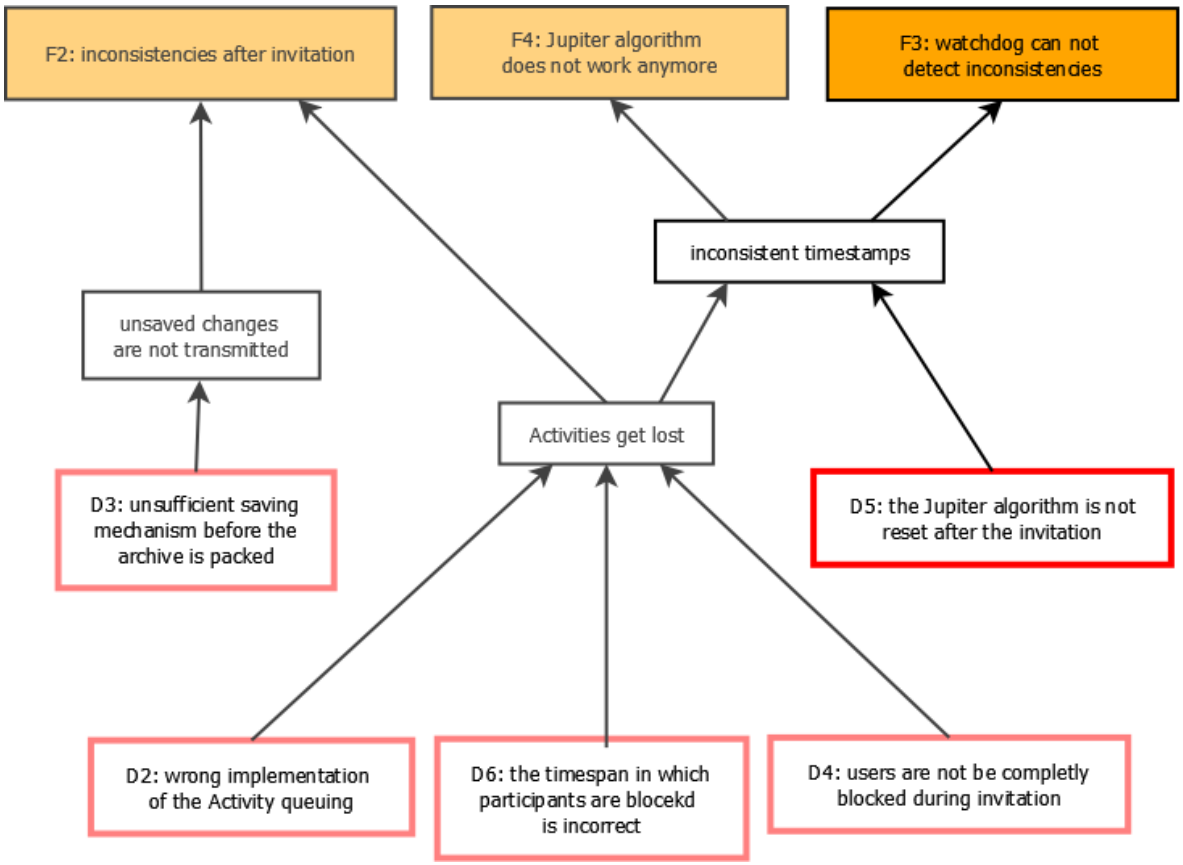

Figure 9: Defect Overview V

[12] http://saros-build.imp.fu-berlin.de/gerrit/#/c/442/

Step 8

The last step in my implementation is to fix the Needs-Based Synchronization. As described earlier this is an optional feature that can be disabled in the Saros settings. The changes described in the previous steps only prevent inconsistencies if this feature is disabled. It seems that when this feature was implemented the use case that participants can be added during a Session was neglected and not tested. As described in section 4.5 this feature consists of Pre-Sharing and Quick-Adding. I had to remove the Pre-Sharing feature, because it was causing the described failure F1. The Quick-Adding feature does not impair the invitation process.

5.4 Identifying Root Causes

In the previous section I described six different defects that are the causes for the failures F1...F4. The next step would be to analyze how and when these defects were introduced as well as when they first caused any of the described failures. Since analyzing the defects, especially the ones related to the Activity queuing problem, took too much time I was not able to identify the root causes. I will continue to work on this and publish the results on my thesis website (`https://www.mi.fu-berlin.de/w/SE/ThesisDPPConsistencyRCA`).

6 Conclusion

The main goal of this thesis was to improve the reliability of Saros concerning the aspect consistency. I was able to partly reach this goal. As described in section 5.3 I was able to fix four of the six identified defects. The patch concerning the remaining two defects is currently in the review process. Although my implementation could increase the reliability of Saros I also had to decrease the functionality. I disabled the Pre-Sharing feature and instead of implementing the invitation process without the blocking of all participants I had to increase the period in which the users are blocked during the invitation process.

A secondary goal was to perform a Root Cause Analysis to identify and fix root causes on the product, and if possible, on the process level. I successfully transfered the basic concepts of the RCA to analyze Saros but I haven't finished the identification of the root causes, yet. The reasons for this are the fact that some of the defects were difficult to understand and to fix and therefor I spent about twice as much time as I had planned for the data collection and defect correction. This leads to the second problem: insufficient time management. I should have aborted the implementation of the Activity queuing much earlier in order to have more time for the identification of root cases, but on the other hand I would have performed the identification with less data.

All in all I plan to finish the RCA and I will publish the results on my thesis homepage. I hope that way I can further improve the reliability of Saros. There is also the possibility that after resolving the root causes the implementation of a blocking-free invitation process might be possible.

7 Appendix

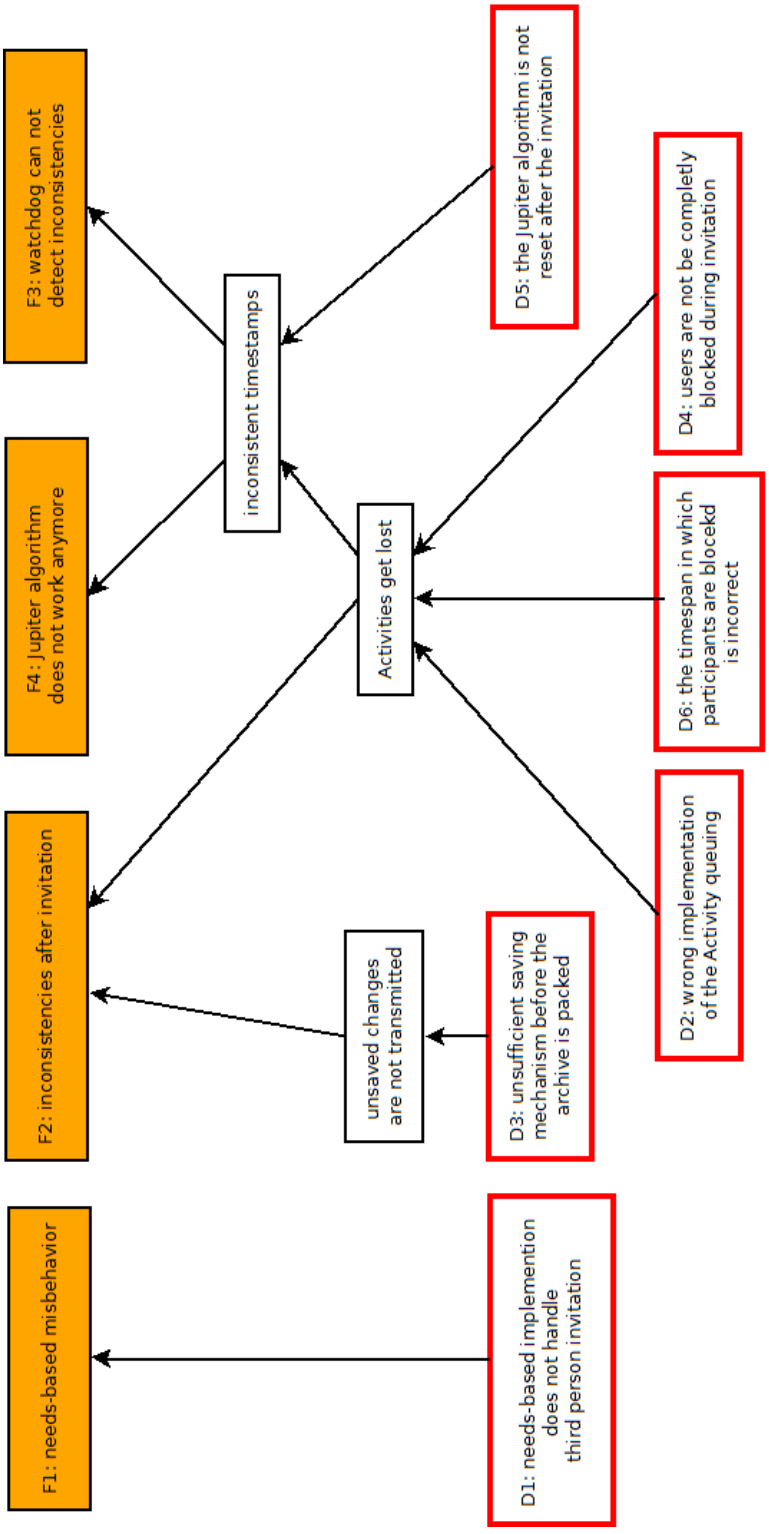

Figure 10: A1 defect overview

References

[Dje06] Riad Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung. Master's thesis, Freie Universität Berlin, 2006.

[Doh11] Christian Dohnert. Unterstützung mehrerer Projekte in einer Saros-Sitzung. *Freie Universität Berlin*, 2011.

[JJR04] Lee N. Vanden James J. Rooney. Root Cause Analysis For Beginners. https://servicelink.pinnacol.com/pinnacol_docs/lp/cdrom_web/safety/management/accident_investigation/Root_Cause.pdf, 2004.

[Sar11] SarosTeam. Invitation. http://www.saros-project.org/Invitation, 2011.

[SS00] Till Schümmer and Jan Schümmer. Support for distributed teams in extreme programming. In *Proceedings of eXtreme Programming and Flexible Processes Software Engineering - XP2000*, pages 355–377. Addison Wesley, 2000.

[Sót09] Tas Sóti. Einladungsprozess in Saros. *Freie Universität Berlin*, 2009.