Master's Thesis at the Institute for Computer Science of the Freie Universität Berlin

Research Group Software Engineering

Designing and Implementing IDE Cross-Compatibility for Saros

Tobias Bouschen
Student ID: 4676966
tobias.bouschen@fu-berlin.de

Advisor:	Kelvin Glaß
First Examiner:	Prof. Dr. Lutz Prechelt
Second Examiner:	Prof. Dr. Claudia Müller-Birn

Berlin, November 6, 2020

Abstract

Saros is a plugin for integrated development environments (IDEs) designed for distributed pair programming. Its goal is to provide the technical infrastructure necessary for developers to concurrently collaborate on shared resources without being co-located. Furthermore, it provides the benefit of allowing the users to stay in their usual workspace using their own configurations, thereby avoiding the potential loss of productivity due to different workspace setups.

Saros was initially only designed for Eclipse but was later on extended to support other IDEs, currently mainly IntelliJ IDEA. The introduction of Saros implementations for other IDEs broadened the usability and potential user base but also led to new difficulties: The Saros implementations for different IDEs were not compatible with each other. Providing such IDE cross-compatibility would bridge the gap between the users of different IDEs, allowing them to still use their preferred IDE during distributed pair programming sessions independent of the IDEs used by other participants. This would further broaden the potential user base and use cases for Saros.

The goal of this thesis is to provide such an IDE cross-compatibility for Saros in general and the current Saros implementations for Eclipse and IntelliJ IDEA in particular.

To reach this goal, I analyzed the current state of the Saros plugin, identified a list of roadblocks that need to be resolved to reach IDE cross-compatibility, and worked on resolving the major roadblocks.

With this thesis, I achieved basic IDE cross-compatibility between Saros for Eclipse and for IntelliJ IDEA. Furthermore, I laid out a plan on how to improve this basic IDE cross-compatibility, mainly concerning improvements to the usability of Saros in cross-IDE setups.

While there are still some remaining restrictions and issues regarding IDE cross-compatibility that were not resolved due to time restrictions, I nevertheless identified and resolved the major roadblocks and provided a clear path on how to expand and improve IDE cross-compatibility in the future.

Eidesstattliche Erklärung

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.

November 6, 2020

Tobias Bouschen

Contents

I Introduction

Pair programming (PP) is a practice most commonly associated with agile development and extreme programming. It consists of two developers working cooperatively on the same task. This cooperation can help reduce the number of faults in the developed software and improve the general design and quality of the created software while only causing a small to moderate overhead in development time and cost. Due to the cooperative work, it can improve the relationship between the developers, accommodate knowledge transfer, and lead to a broader sense of general shared code ownership. [1]

The exact type of cooperation can have many forms, but, most commonly, one developer takes the role of the "driver" and the other the role of the "navigator". The "driver" has control of the keyboard and works on the specific problem or task at hand. The "navigator" reviews the created code on-the-go and keeps an overview of the larger issues or tasks and the next steps to take. [2]

With the growth of the IT sector and the rise in globalization and outsourcing, developers are commonly spread across different locations, making traditional pair programming unfeasible. To resolve the issue of co-location, distributed pair programming (DPP) tries to replace the local cooperation through technical means. It does so by allowing shared access to the worked-on resources and, more importantly, by providing the necessary awareness information that would otherwise be provided by the shared workspace. As an added benefit, it allows the users to work collaboratively while still remaining in their familiar development environment, thereby avoiding the potential loss of productivity due to different workspace setups between developers (e.g. different keyboards, keyboard layouts, operating systems, software, etc.). [3]

Saros[1] is a plugin for integrated development environments (IDEs) providing the technical infrastructure for distributed pair programming. It allows sharing source code with other users and concurrently working on the shared code. Additionally, Saros provides the workspace awareness information[2] necessary for tight collaboration, showing what other users currently are doing in the shared workspace. [3]

Saros was initially created for the Eclipse IDE[3] but was later expanded to support other IDEs, currently IntelliJ IDEA[4] and other JetBrains IDEs. This made the plugin usable by a wider range of developers but introduced a new issue: The different implementations of the Saros plugin were not compatible with each other. This was mainly due to design issues in the logic shared by the IDE implementations and

[1] https://www.saros-project.org/
[2] https://www.saros-project.org/documentation/awareness-information.html
[3] https://www.eclipse.org/
[4] https://www.jetbrains.com/idea/

difficulties bridging the gap created by technical differences in the underlying models for the different IDEs.

Providing such IDE cross-compatibility would further increase the benefit of allowing the users to work in their usual environment by bridging the gap between different IDE users, thereby further broadening the potential user base and use cases for Saros. Furthermore, with the range of other plugins supporting distributed pair programming out there [3] and some IDEs even providing their own, native solutions for distributed collaboration — examples being Visual Studio's Live Share[5] and the in-development CodeWithMe plugin for JetBrains IDEs[6] — supporting IDE cross-compatibility would offer a unique selling point to attract new potential users and differentiate Saros from its competitors.

1 Goal of This Thesis

The main goal of this thesis is providing IDE cross-compatibility capabilities for the Saros plugin. To make it more approachable, I split it up into the following sub-goals:

1. Analyze the current state of the Saros plugin and its IDE implementations.
2. Identify potential roadblocks on the way to supporting IDE cross-compatibility.
3. Resolve the necessary roadblocks.

2 Structure of This Thesis

This thesis is split into five parts:

I Introduction gives a short introduction of the thesis topic and presents the goal of the thesis. (You are currently reading this part.)

II Fundamentals provides context for the thesis by giving some fundamental information related to the thesis topic as well as references to related work.

III Implementation gives an overview of the work done as part of the thesis, describing the process and the implementation results.

IV Conclusion lists the topics and issues not resolved as part of the thesis, evaluates the thesis results, and provides suggestions for future work.

V Appendix contains a list of all pull requests created as part of this thesis as well as some additional figures.

A more detailed description of the structure is given at the start of each of the following parts.

[5] https://code.visualstudio.com/blogs/2017/11/15/live-share
[6] https://plugins.jetbrains.com/plugin/14896-code-with-me

II Fundamentals

This part offers some fundamentals to provide context for the rest of the thesis. Section 3 gives a short overview of Saros, its functionalities, and its development history in order to provide some information on the current state of Saros and how it got there. Afterwards, section 4 describes the algorithm used for consistency management in Saros to provide the necessary context to understand the work done on these components. Lastly, section 5 gives an overview of related work, mainly focusing on the theses that laid the groundwork for this thesis.

3 Saros

Saros is an IDE plugin written in Java aimed at supporting distributed pair programming. It is developed as an open source project[7] with large parts of the work being contributed by the working group Software Engineering[8] at the Freie Universität Berlin, either through (student) research assistants or student theses. It is currently available for Eclipse (named Saros/E) and IntelliJ IDEA (named Saros/I). Furthermore, a stand-alone server component named Saros Server is in development which can be used to independently host Saros sessions.

Saros provides the capabilities to collaboratively work on source code by synchronizing the workspaces of all participants, allowing for concurrent edits of the shared source code. It also provides the workspace awareness information necessary for tight collaboration[9]. It shows which participants are part of the current session and which editor they have currently active. In each shared editor, it also shows a range of annotations providing additional awareness information:

Contribution Annotations highlighting the most recent changes made by each other participant.

Cursor Annotations highlighting the current cursor position of each other participant.

Selection Annotations highlighting the current selection of each other participant.

Viewport Annotations highlighting the currently visible part of the editor for each other participant.

Furthermore, Saros offers a follow mode meant to support the standard pair programming setup of a driver and a navigator. When in follow mode, the local IDE view will automatically be adjusted to match the view of the followed participant. If

[7] https://github.com/saros-project/saros
[8] https://www.mi.fu-berlin.de/inf/groups/ag-se/index.html
[9] https://www.saros-project.org/documentation/awareness-information.html

the followed participant, for example, scrolls or changes the editor, the local state of the following participants will be adjusted accordingly.

3.1 History of the Development of Saros

The initial version of Saros was created in 2006 by R. Djemili as part of his diploma thesis [4]. It was first designed with a much smaller feature set, allowing only two participants and enforcing a strict separation between the driver and navigator role. This scope was later extended by many other features (like an increased number of participants, concurrent text modification, etc.) through different theses and other work in the working group Software Engineering (see section 5.1).

While the development through student theses ensured continual development of the Saros plugin, the need to provide topics for new theses led to constant feature creep and the high fluctuation of developers — many only working on the plugin for their thesis — took its toll on the code quality. Furthermore, most developers were relatively inexperienced — Saros being the first major software project they were working on — and the tight schedule of a thesis often did not allow for the necessary time to get properly familiarized with the vast and complex code base. The impact of this high developer fluctuation was somewhat reduced by introducing continuous integration and an improved testing suite [5] as well as static code and architecture analysis [6] in combination with requiring code reviews, but the code quality was still subpar and the error density still quite high. The code quality has increased recently due to the reduction of student theses, the removal of unnecessary or unmaintainable features, and the continued work of a relatively stable core team.

In its initial stages, the plugin was only designed for Eclipse. To make its development as fast and easy as possible, the plugin was tightly integrated with the Eclipse API, leading to many designs and interfaces mirroring the Eclipse counterparts. Later on, the decision was made to extend Saros to support other IDEs as well. IntelliJ IDEA was chosen as the best candidate due to its rising popularity among Java developers. First, an initial Saros version for IntelliJ IDEA was created by copying a lot of the Eclipse-specific code and adjusting it for the integration with IntelliJ IDEA. This code duplication led to issues with the maintainability of Saros. Saros for Eclipse was still the main focus of the development efforts, leading to many changes not being applied to the code duplicates in the IntelliJ IDEA implementation. As a result, the code duplicates drifted apart in content, making it harder to unify them later on.

To avoid such duplication and slim down the code base, the core business logic of the plugin was extracted into a separate component, the Saros Core[10] (see section 5.2). This process mainly concerned moving logic that was deemed IDE-independent but did not evaluate the architecture itself. As a result, many of the classes and interfaces created in the Saros Core as part of this refactoring were still heavily influenced by the Eclipse API they were initially designed for, often offering almost identical signatures.

[10]The Saros Core component was actually created before the work on the initial Saros version for IntelliJ IDEA started. But, in its initial stages, the Saros Core project remained largely empty, offering only a very limited feature set. The actual extraction of significant business logic only took place after the initial Saros/I prototype had already been created.

This proved to be very cumbersome when trying to cleanly implement Saros for other IDEs (including IntelliJ IDEA) as the central design concepts used by the Eclipse IDE do not always apply to other IDEs [7].

4 Jupiter Algorithm

As mentioned in the previous section, Saros supports the concurrent editing of shared documents. To do so, it needs to ensure that the contents of the shared documents stay synchronized for all participants. This is done using the Jupiter algorithm, named after the Jupiter System it was modeled after, which was proposed in *High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System* by Nichols, Curtis, Dixon, and Lumping in 1995 [8].

The algorithm uses a centralized coordinator architecture, meaning all clients report their local changes to a central host which then dispatches them to all other clients. As a result, the host is used as a centralized synchronization point. This allows for the network to be modeled as a set of independent host-client-connections.

To ensure that all participants are working on the same document contents, synchronous changes that influence each other have to be applied correctly on all sides. The Jupiter algorithm uses optimistic concurrency control, meaning all changes are directly applied locally and then sent to the other participants. As other participants might have made conflicting changes in the meantime, received changes must be transformed first before they can be applied locally. This is done through the operational transformation algorithm proposed in *Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems* by Sun, Jia, Zhang, Yang, and Chen in 1998 [9].

The goal of the algorithm is to guarantee that the following three properties are always maintained:

Convergence After all activities were applied on all sides, the document contents match for all participants. In particular, this means that temporary differences in the document contents are acceptable as long as not all activities have been applied locally.

Intention Preservation The execution of an activity has the same effect as though it were applied to the state that it was created with. In particular, this means that received changes which modify content that was also modified locally in the meantime have to be adjusted accordingly.

Causality Preservation For each participant, an activity is only applied after all other activities it is dependent on have been applied. An activity A_1 is dependent on another activity A_2 if and only if A_1 was created based on a local state that had already applied A_2, either because A_2 was created locally or had been received and applied previously.

This property is not as important for Saros as, with the centralized host architecture, it can be guaranteed simply by ensuring that all activities are delivered

and applied in the order they were sent.

In general, the operational transformation algorithm deals with adjusting received activities for concurrent local changes by modifying the offset or content of the activity accordingly. The need to adjust activities can be shown with this simple example: With the initial state of "`Hello world`", participant P1 inserts "`my `" at offset 5 and participant P2 concurrently inserts "`!`" at offset 11. The intended result is "`Hello my world!`".

For P2, this already works out by default as the local change is located behind the remote change:

Initial state: "`Hello world`"

P2: ins("`!`", 11) $\Longrightarrow$ "`Hello world!`"

P1: ins("`my `", 5) $\Longrightarrow$ "`Hello my world!`"

For P1, however, an adjustment of the received activity is necessary. Otherwise, the intention of the received activity is not preserved and the document contents no longer converge:

Initial state: "`Hello world`"

P1: ins("`my `", 5) $\Longrightarrow$ "`Hello my world`"

P2: ins("`!`", 11) $\Longrightarrow$ "`Hello my wo!rld`"

To correct the received activity of P2, the index must be shifted back by the length of the local insertion done by P1. This results in the adjusted activity of P2 inserting "`!`" at position 14. Applying this adjusted activity instead causes the document contents to converge again.

Initial state: "`Hello world`"

P1: ins("`my `", 5) $\Longrightarrow$ "`Hello my world`"

P2: ins("`!`", 14) $\Longrightarrow$ "`Hello my world!`"

These transformations get much more complex when the state of the different participants diverges by more than one change. In such cases, the received changes have to be transformed in relation to all local changes not yet applied to the sender side, requiring a separate transformation for each such local change.

5 Related Work

5.1 Saros in General

Saros was initially designed and created as part of the diploma thesis *Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung* (R. Djemili, 2006) [4] (roughly translates to *Developing an Eclipse-Extension to Realize and Record Distributed Pair Programming*). It was extended and improved as part of many follow-up theses.

A general overview of the field of agile development and pair programming as well as Saros, its development history, how it compares to other, related tools, and a detailed analysis of the design of Saros measured under the viewpoint of usability in an industrial setting is given in the dissertation *Industrially Usable Distributed Pair Programming* (J. Schenk, 2018) [3].

5.2 Saros Core Extraction

The groundwork to make Saros IDE-independent was laid by previous theses extracting the IDE-independent logic into what is now called the Saros Core. This process was started with the bachelor's thesis *Refaktorisierung des Eclipse-Plugins Saros für die Portierung auf andere IDEs* (A. Lasarzik, 2015) [10] (roughly translates to *Refactoring of the Eclipse Plugin Saros to Facilitate Porting to Other IDEs*) and continued with the master's thesis *Entwicklung und Evaluation eines unabhängigen Sitzungsservers für das Saros-Projekt* (D. Washington, 2016) [11] (roughly translates to *Development and Evaluation of an Standalone Session Server for the Saros Project*) and the bachelor's thesis *Verbesserung und Erweiterung der Core-Bestandteile von Saros* (D. Sungaila, 2016) [12] (roughly translates to *Improvement and Extension of Parts of the Saros Core*).

While these theses did not complete the extraction of the shared logic, their work was essential for the further development of the Saros plugin and its porting to other IDEs.

5.3 Saros for IntelliJ IDEA

Saros for IntelliJ IDEA was started through the work of an independent contractor working together with the Saros project. This work was continued with the bachelor's thesis *Bestandsaufnahme und Arbeit an einer Alpha-Version des Saros-Plugins für die IntelliJ-Plattform* (T. Bouschen, 2017) [7] (roughly translates to *Surveying and Working on an Alpha-Version of the Saros Plugin for the IntelliJ Platform*). With the continued work that followed after the thesis, the Saros plugin for IntelliJ IDEA was finally in a state where early alpha versions could be released.

The plugin being in a usable state, even with just a minimal set of functionalities, was almost a necessity for this thesis as it allowed for a better analysis of potential problems with IDE cross-compatibility.

5.4 Saros Filesystem Rework

Prior to this thesis, the master's thesis *Saros: Refactoring the Filesystem* (O. Hanßen, 2019) [13] already looked at improving the filesystem abstraction used in the Saros Core. The main goal of these changes was making the logic more IDE-independent, thereby making it easier to adapt Saros for IDEs with different workspace models. While the concrete approach or implementation of the thesis was not used, it provided a general jumping-off point for the rework of the filesystem implementations done as part of this thesis.

III Implementation

This part describes the process and results of the thesis. It starts with section 6 giving an overview of my general approach. This general approach is then expanded upon in the following sections, with section 7 describing my preparations, section 8 describing the analysis of the requirements necessary to reach the defined goal, and section 9 describing the roadblocks identified as part of this requirement analysis. Afterwards, it describes the actual implementation work done as part of the thesis, starting with section 10 describing the implementation of line ending normalization, followed by section 11 describing the preparations for the filesystem rework as well as section 12 describing the implementation of the filesystem rework, and ending with section 13 describing various other issues regarding IDE cross-compatibility that were also resolved as part of this thesis.

In general, when describing implementation work done as part of this thesis, some of the more technical details are skipped for the sake of brevity. Many of such details require deeper technical knowledge of Saros and its IDE implementations to be helpful. Explaining them and the background knowledge they rely on would substantially increase the scope of this thesis while contributing little to the understanding of the main issues. If such details are still of interest, the pull requests created as part of this thesis are always referenced in the corresponding sections. These pull requests contain the exact code changes that were made and the discussions that had been had during the development process. Furthermore, a list of all pull requests created as part of this thesis, including some minor changes not explicitly mentioned in this part, is given in appendix A.

6 General Approach

To approach the topic, I started by reading up on Saros literature in order to gain a better understanding of the base design concepts behind Saros. Furthermore, I wanted to check whether the current feature set of Saros still matches the requirements for a distributed collaboration tool.

Afterwards, I analyzed the existing Saros code to identify possible roadblocks that stand in the way of IDE cross-compatibility.

Then, for each of the identified roadblocks, I gauged the severity, suggested possible solution approaches, and discussed the benefits and drawbacks of each suggested approach. Furthermore, I created a list of preliminary steps that most likely would have to be taken to resolve the roadblock, allowing me to better plan the implementation process. When proposing adjustments for existing interfaces in the Saros Core or new models for some of the general Saros functionality as part of these approaches, I had

to make sure that my knowledge of the Saros/I implementation as well as the IntelliJ IDEA IDE architecture did not influence the suggested designs. Otherwise, the resulting architecture could suffer from the same issues as the previous one, namely being too closely modeled after a specific IDE model.

After this list of steps was finalized, I started implementing the necessary steps to reach a base level of IDE cross-compatibility. During this process, I again had to make sure that the created Core logic was not influenced by models specific to the Eclipse or IntelliJ IDEA architecture.

As it was unlikely that I was going to be able to implement all steps to begin with, I chose the steps to implement based on their necessity and on their difficulty. This led to an agile process where the next goals were decided ad hoc and some steps were pushed back or completely dropped from the scope of the thesis as they were found to be less crucial than they first appeared. During this implementation process, I regularly discussed the current work as well as how best to proceed with other members of the Saros team as part of the twice-weekly stand-ups.

All patches created as part of this thesis were submitted as GitHub pull requests and peer-reviewed by other members of the Saros project. To make the review process as easy as possible, I broke down my patches into multiple commits wherever possible, grouping the changes by topic. Especially for larger patches, this makes the changes more approachable and allows for a more structured review process. It enables the reviewer to do the code review in chunks, distributing the effort required for the lengthy review process.

When creating patches, I paid special attention to providing detailed documentation for my code and descriptive messages for my commits. I also tried to follow the boy scout rule[11] wherever possible, including incremental code cleanups with my changes, thereby improving the general code quality during the process. This should make the code easier to maintain and work with in the future, hopefully resulting in a lower rate of code decay than seen in other parts of the Saros codebase. Furthermore, during the development process, I tried to make the thoughts behind my work and the next steps I was going to take very clear to make interactions with other members of the Saros team as easy as possible.

7 Preparation

To get a better understanding of the Saros features, their purpose, and the design decisions behind them, I read the base literature on the Saros design and history ([3, 14–17]). This base knowledge is essential when having to make the decision whether or not to cut a feature to allow for (easier) IDE cross-compatibility, as reducing the usability of Saros should largely be avoided. A short analysis of the Saros feature set in order to determine whether it still matches the requirements of a distributed collaboration tool did not reveal any obvious shortcomings. Saros offers the main functionality needed for collaborative editing of documents as well as providing ade-

[11] https://clean-code-developer.com/grades/grade-1-red/#Boy_Scout_Rule

quate workspace awareness [3]. While it still provides additional features that might be seen as superfluous, e.g. the direct transfer of arbitrary files between participants, none of them provided issues for IDE cross-compatibility.

Afterwards, to get a better understanding of the previous efforts to make Saros more IDE-independent, I read past theses on the topic of the Core extraction ([10–12]) and previous work on the filesystem abstraction rework ([13]).

On the technical side, not very much familiarization with the code base was necessary. I had already been working in the Saros project as part of my bachelor's thesis ([7]) and continued this work as part of my subsequent employment as a student researcher in the working group Software Engineering at the Freie Universität Berlin. Due to this work, I was very familiar with the IntelliJ IDEA implementation of Saros and also had a good understanding of the other Saros components.

8 Requirement Analysis

To gain a general understanding of the challenges that would face me as part of this thesis, I started by analyzing the current Saros implementation by reading the old architecture documentation[12]. Even though it is somewhat out of date, it still provided a good overview.

Afterwards, I tried gaining a more complete understanding by skimming the Saros code base, paying closer attention to the central classes of the Saros Core and the non-UI-related parts of the Eclipse and IntelliJ IDEA implementations. During this process, I was specifically looking for (implicit) technical assumptions that were made to match the design of a specific IDE but might not apply to other IDEs.

I also had a look at other currently popular Java IDEs (mainly Visual Studio Code[13]) to look for further general architectures and restrictions to keep in mind. This, however, proved to be difficult as it quickly became apparent that most issues with IDE cross-compatibility are due to subtle differences in the technical implementations of the IDEs. Finding such differences takes a lot of time and knowledge of said implementations, making it unfeasible as part of this thesis.

I then created a list of potential roadblocks that had to be resolved on the way to providing IDE cross-compatibility. This list was in no way exhaustive as many issues in the context of cross-compatibility are based on very specific technical details that are hard to identify. Furthermore, not all of the roadblocks were of the same severity. Some had to be resolved for basic IDE cross-compatibility while others only had to be resolved to offer the optimal user experience.

For each of the found roadblocks, I proposed, compared, and evaluated different solution approaches.

[12] https://saros-project.github.io/legacy_docs/architectureDocumentation.html
[13] https://code.visualstudio.com/

9 Found Roadblocks

I identified the following roadblocks:

1. Issues with the Handling of Line Endings
2. Issues with the Current Filesystem Implementation
3. Issues with Determining Which Resources Not to Share

Some of these roadblocks were found as part of the explicit requirement analysis, some had already presented themselves during my previous work on Saros/I, and some were discovered during the planning and development process of the thesis due to new insights into potential problems and newfound restrictions of the proposed solution approaches.

The initial list of roadblocks also contained an issue regarding the handling of binary files. To work around differences in the handling of binary files in Eclipse and IntelliJ IDEA, I suggested introducing a strict separation between binary and non-binary content for Saros. As the Eclipse API did not provide an adequate method to differentiate between binary and non-binary (i.e. text) files, I needed a different way to check for binary files. But I was neither able to find a good metric for deciding whether a file is binary nor could I find a library providing such a functionality. Ultimately, when reevaluating the roadblock in the context of IDE cross-compatibility at the start of the implementation phase, it no longer appeared as pressing as first thought. As a result, I decided to drop the roadblock, only keeping a couple of key issues related to IDE cross-compatibility around as miscellaneous tasks. I resolved these important aspects of the dropped roadblock as described in section 13.1.

As part of this thesis, I only managed to resolve roadblocks 1 and 2 due to time constraints. I still proposed and evaluated solution approaches for the unresolved roadblock. They can be found in section 14.1.

9.1 Issues with the Handling of Line Endings

With the current operating systems, there are two different ways of signaling the ending of a line in a text file: "\n" for UNIX-based operating systems and "\r\n" for the Windows operating system.

In Eclipse, like in most IDEs and editors, the line endings written in the file are also used when processing it internally or representing it in the editor. In IntelliJ IDEA, however, line endings are normalized to UNIX line endings internally[14]. All line endings in the file are substituted with the UNIX line ending when it is read from disk and the original line endings are then re-substituted when the content is written back to disk.

This poses a big issue for IDE cross-compatibility as, even when working on the same content on disk, the document content available in the IDE does not have to match. In particular, when trying to maintain a session between an IntelliJ IDEA instance

[14] https://jetbrains.org/intellij/sdk/docs/basics/architectural_overview/documents.html

and an Eclipse instance while sharing files containing Windows line endings, Saros will always detect an inconsistency in the shared files as the line endings will not match. Furthermore, as the two line separators are not of equal length, the UNIX line separator consisting of one character and the Windows line separator consisting of two characters, the character-based offsets will be different in the two files. This poses another challenge for Saros, which relies heavily on such offsets to apply remote changes correctly and display awareness information such as remote cursor locations and selections.

9.2 Issues with the Current Filesystem Implementation

Different IDEs use different models to represent the structure of the aggregates of files representing a piece of software. As Saros wants to interact with these files, it needs to be able to interface with this logic.

Since Saros was first only designed for Eclipse, the interface to interact with the filesystem logic matches the Eclipse counterpart, going so far as to simply refer to the Eclipse documentation if no other explicit documentation was given for a method or class. The design was also not amended when extracting the logic to the Saros Core, meaning it was still very much based on the Eclipse counterpart. This makes the implementation of Saros for other IDEs more cumbersome as the IDE-specific filesystem interface has to be somehow mapped onto the Saros (or rather Eclipse) filesystem interface. Even when such a mapping is possible, the underlying implementation still has to be IDE specific to match the IDE filesystem model. This can lead to a confusing mix of names and concepts in the implementation, especially when naming schemes clash in the different IDE concepts, an example being the usage of the term "project" in the Eclipse and IntelliJ IDEA model[15].

Though this issue with mapping new IDE implementations onto the Core interface is cumbersome when implementing Saros for new IDEs, it does not provide as much of an issue for IDE cross-compatibility. A more pressing issue is that the current sharing logic uses IDE-specific concepts to determine the files to share with other participants. Saros for Eclipse allows the user to share projects[16] while Saros for IntelliJ IDEA allows the user to share modules[17]. While these concepts are generally comparable in scope, it is not always possible to create a clean mapping between Eclipse projects and IntelliJ IDEA modules for the same piece of software as the filesystem models of Eclipse and IntelliJ IDEA are far too different. This especially becomes an issue when trying to create a project/module as part of a cross-IDE Saros session, as the other IDE models won't necessarily provide all the information needed to correctly represent it in the local model. These complications grow with the complexity of the shared project/module setup.

As an example, even with a relatively normal setup, like the Saros project itself, a map-

[15] https://www.jetbrains.com/help/idea/migrating-from-eclipse-to-intellij-idea.html#term

[16] https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-12.htm

[17] https://www.jetbrains.com/help/idea/creating-and-managing-modules.html

ping between Eclipse projects and IntelliJ IDEA modules is far from trivial: Eclipse sees each Saros component (e.g. Core, Eclipse, IntelliJ) as one project. IntelliJ IDEA, on the other hand, imports every Saros component through Gradle[18] as three modules, each containing different parts of the tree: a base module containing the general resource tree, a "main" sub-module containing only the main sources and resources, and a "test" sub-module containing only the test sources and resources. As a result, there is no clean mapping between the Eclipse and IntelliJ IDEA model as the same resources that are represented by one project in Eclipse are spread across three modules in IntelliJ IDEA. Furthermore, some of these IntelliJ IDEA modules can't be represented as a unified tree (without including parents outside the module) as they are the combination of different resource trees. There is no way of representing such a collection of resources as a single project in Eclipse.

9.3 Issues with Determining Which Resources Not to Share

Generally, projects/modules may contain some resources that should not be shared through Saros, examples being automatically generated artifacts or files containing user specific data. A good definition for most of these files would be "files that should not be kept under versioning", i.e. that should be ignored by the used version control system (e.g. Git, Subversion, etc.).

The current approach of Saros is to use local IDE mechanisms to determine which resources not to share. Most IDEs offer the option to ignore resources like build artifacts in the local workspace in order to exclude them from functionalities like inspections, searches, and refactoring tools. Eclipse offers the concept of derived resources[19] and IntelliJ IDEA the concept of excluded resources[20] for this purpose.

The issue with this approach is that, in both cases, these concepts are generally only used to mark resources that are generated as part of the compilation process. However, there also are other automatically generated resources and resources that contain user specific data that should not be shared between Saros instances. When implementing Saros to only work for one specific IDE, it is quite simple to cover most of these special cases with dedicated logic to exclude such resources as most of them can be queried from the IDE. But this only works if the set of non-shared resources determined through the IDE specification is the same for all participants. If participants use different IDEs, the set of such additional non-shared resources might vary between the participants, leading to a perceived desynchronization as Saros detects files for some participants while they are ignored for others.

10 Implementing Line Ending Normalization

The first roadblock I decided to address was the handling of line endings described in section 9.1. I made this decision as resolving the issue seemed essential to allow-

[18] https://gradle.org/

[19] https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_derived.htm

[20] https://www.jetbrains.com/help/idea/content-roots.html#folder-categories

ing IDE cross-compatibility. Furthermore, the issue was already well-defined and confined to a specific section of the code, meaning it was easier to approach.

As an alternative, I also considered resolving the issue with the filesystem implementation described in section 9.2 first since resolving it promised a potentially larger gain in compatible functionality. Furthermore, resolving this roadblock would remove a more substantial issue with the Saros design, also providing a large benefit for the general Saros development in the future. But I was hesitant to start with the roadblock as I was not certain how much effort it would require. The filesystem implementation interacts with many other Saros components, meaning reworking it would also require reworking or at least adjusting many related components. This especially applies to the IDE-specific UI implementations, whose adjustment and/or redesign can take up an almost unlimited amount of time depending on how much attention is paid to usability. So, as the resolution of either of the roadblocks seemed essential to enabling IDE cross-compatibility, I decided resolving the line ending issue first was the better approach to ensure that I could resolve as many important roadblocks as possible in the time I had.

10.1 Solution Approach

The most straightforward solution for this problem would have been requiring the usage of the UNIX line ending for all shared files. This is, however, not always realistic as some files might require a specific line ending. Furthermore, this would go against the default Git workflow, in which the local line endings are substituted when checking out a project and the UNIX line endings are re-substituted when checking in new code[21], meaning it could substantially reduce the ease of use of Saros by requiring the user to change the setup of their local development environment.

As a more general solution approach, I decided to introduce line ending normalization for Saros. This means that Saros substitutes all line endings when reading content from an editor and re-substitutes the initial line endings when writing changes to the editor. Normalizing the content ensures that all participants are working on exactly the same content, independently of the line ending handling of the local system or IDE. Ensuring a unified content handling is crucial for Saros to function, as it is otherwise much harder to correctly apply remote actions to the local workspace and to correctly validate that the workspace state of all participants is still in sync. Files containing mixed line endings are explicitly not supported by the normalization logic.

But, even when normalizing the line endings internally, the issue of the differing offsets still remains. The offsets are obtained through IDE-specific APIs. They are read from the potentially non-normalized content and would therefore still have to be adjusted. Such corrections could be done by counting the line separators from the start of the file up to the given offset and then adjusting the value accordingly depending on the line ending substitution. While this approach does work, it does have the drawback of requiring Saros to go through the entire file content up to the offset every time, causing a non-negligible overhead. As the offset is required for

[21] https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration#_core_autocrlf

basically every action transferred by Saros, this overhead could be quite substantial, potentially making Saros unusable. This overhead could be reduced by introducing a caching mechanism for quicker offset corrections, but this cache would be difficult to implement and would have to be accurately updated for every document change.

To avoid having to calculate the offset correction, I instead decided to replace the current position handling with a line-ending agnostic one. More specifically, I decided to implement a positioning system that uses the line number and in-line offset to specify a position in a document. With such a positioning scheme, it is possible to specify a position in a document independently of the contained line endings as the character-based offset is only used to specify the position inside a specific line. Choosing this approach also meant that I had to adjust the Jupiter algorithm since it uses the text position as part of the operational transformation.

This approach still requires Saros to calculate the line number and in-line offset. While the in-line offset is a simple calculation given the offset of the line start, calculating the corresponding line number for a specific offset is more complex. In the worst case scenario, this requires the same computational effort as correcting the character-based offset, having to count the line endings up to the given offset. But, while the caching logic to speed up the offset correction logic would have to be implemented from scratch, most IDEs, including Eclipse and IntelliJ IDEA, already offer an optimized API that can convert between offsets and line numbers. This makes this approach much more favorable compared to correcting the character-based offsets, even though it requires a larger rework of the Saros logic.

To resolve the roadblock, I started by reworking the text positioning scheme used by Saros. Introducing line ending normalization first did not seem sensible. Much of the Saros logic relies on correct text position handling. The previous, character-based offset handling breaks when operating on normalized content as the used offsets are obtained through the IDE API. This IDE API calculates such offsets using the internal, non-normalized content. As a result, the offsets can't be used for normalized content as they might not point to the correct text position. This would make testing the implemented changes almost impossible until a stable state had been reached, effectively making testing unfeasible until after the text positioning scheme had been reworked as well. Starting with the text positioning logic, then moving on to the Jupiter algorithm, and only then introducing the actual line ending normalization allowed me to work in incremental steps that were still testable throughout the process.

10.2 Reworking the Text Positioning Scheme

I reworked the text positioning scheme in pull request #862 and #872. As already mentioned in the section covering the chosen solution approach, the main point of this rework was to introduce a line-ending agnostic logic to point to a specific location in a document. I chose to implement a new `TextPosition` class for this purpose using the line number and in-line offset to specify a location in a document. Subsequently, I also added a new `TextSelection` class representing a range of text as two `TextPosition` objects.

To support this logic, I also added multiple utility methods to convert between a normal text-based offset and the new positioning scheme. For the Eclipse and IntelliJ IDEA specific part of Saros, I added conversion logic supported by the corresponding IDE API to speed up the conversion process. Next to the IDE specific conversion logic, I also added a general conversion utility located in the Saros Core which can be used by implementations that operate without IDE support, mainly the Saros Server. This general utility is also very helpful for unit testing to mock the logic that would otherwise be handled by the IDE. To ensure that the logic functioned as intended, I also added unit tests for each conversion utility implementation.

After adding the base logic necessary for the new positioning scheme, I integrated it with the current Saros actions. I started by first adjusting the handling of selection annotations as they are not synchronized using the Jupiter algorithm, meaning they could be adjusted without having to touch the operational transformation logic. This again allowed me to proceed in as small steps as possible while maintaining a testable state for Saros.

I then proceeded with integrating the new text positioning scheme into the logic handling local text edits. As part of these changes, I also had to adjust the activities used to send such text edits to other participants. With the previous logic, the activities contained the starting offset and length of the text edit. These parameters were needed to correctly apply the text edit on the remote side and to correctly transform it using the operational transformation algorithm. With the new logic, using the character-count-based length of the text edit is no longer an option as it is not line-ending agnostic. Instead, I decided to replace it with the line delta and in-line offset delta of the text edit. These values describe the delta caused by the text edit, i.e. how many lines were removed or added and how the in-line offset of the last line of the text edit changed. This approach seemed preferable to sending the end position directly, as these deltas can be adjusted much more easily during the operational transformation. If the end position would be sent instead, it would have to be recalculated every time the operation is transformed, requiring to calculate the deltas anyway. Furthermore, with these deltas, the end position can still easily be calculated when needed.

To make the calculation of these deltas easier, I added a utility class to the Saros Core. I also added unit tests to ensure the conversion logic functioned as intended and also worked correctly for different line endings. Even though the logic will only have to work on UNIX line endings due to the line ending normalization, ensuring that it is still compatible with Windows line endings is still useful. This prevents potential issues in case the logic gets used in a different context or our base assumptions change at some point in the future and the content is no longer normalized or the normalization default changes.

10.3 Adjusting the Jupiter Algorithm Implementation

As a result of modifying the positioning scheme used for text edit activities in Saros, I also had to adjust the operational transformation logic used in the Jupiter algorithm. I adjusted the Jupiter algorithm logic in pull request #872.

The adjustment of the operational transformation algorithm proved to be more time-consuming than I first thought. As the new positioning logic uses multiple values to specify a position, the transformation has to handle a lot more cases. With the previous logic, all transformations were solely reliant on the offsets. With the new logic, it makes a difference whether the two operations to transform are located in the same line or not or whether they have a line delta or not. To make the logic easier to understand, I made sure to simplify it as much as possible by extracting repeated logic into separate methods. I also extended the comments explaining which transformation case a specific part of the logic handles and what the expected outcome is, making it easier to get an overview of the logic. Furthermore, I extended the general documentation of the Jupiter algorithm logic to make it easier to understand, maintain, and change in the future.

After adjusting the Jupiter algorithm, I spent considerable time adjusting, updating, and substantially extending the unit tests covering the logic. As this logic is responsible for the core functionality of Saros and is therefore essential for its usability, making sure that it runs correctly had a very high priority for me. Furthermore, faults in this logic create the worst possible category of failures for the Saros plugin as they potentially cause the destruction of content for the user. To make this outcome as unlikely as possible, I invested a lot of time into ensuring that I did not introduce any faults while adjusting the transformation logic. I did so by first updating the existing test cases and then systematically introducing new test cases to cover all transformations that are now possible. I also made sure to document the setup and expected result for each test case to make it as easy as possible to debug test failures and identify missing test cases. During this process, I also simplified and cleaned up the existing testing frameworks by extracting shared logic, introducing helper and utility classes, and extending the documentation.

To further ensure that the changes to the operational transformation logic did not introduce any subtle issues with longer, more complex chains of transformations, I also added a random test for the operational transformation logic with pull request #874. The test creates a set number of random operations which are randomly distributed among two participants. Initially, these operations are only applied locally for the chosen participant, ensuring that a long chain of operations is queued for both participants. As the locally queued operations have to be taken into account when transforming a remote operation to apply to the local state, this creates very complex transformation setups. After a set number of operations has been created, the queued operations for each participant are transformed and applied to the state of the other participant, thereby synchronizing the document states. After each synchronization, the test checks whether the document states actually converged. The number of operations to create and the number of synchronization rounds can be adjusted at will.

In its current form, this test is only meant to be run manually. As it generates random operations, it is not deterministic, which is why I decided to disable it by default. But, even though the test in its base form is non-deterministic, specific test runs are still reproducible. When the test starts, it prints the used seed for the random function. This seed can be set manually, allowing the test to become deterministic in order to

recreate specific test runs and better debug found issues.

10.4 Adding Line Ending Normalization

After I finished the positioning scheme rework and the associated Jupiter algorithm adjustment, I started to implement the actual line ending normalization logic with pull request #877. For the normalization, I chose the UNIX line separator ("\n") as the default to use internally. I tried to make this decision based on the most likely use cases to minimize the overhead produced by the conversion. Even though a majority of developers still use Windows as their operating system, which is most likely even more prevalent in bigger companies, UNIX based operating systems still also hold a considerable market share[22]. Furthermore, as IntelliJ IDEA already normalizes to UNIX line endings internally, it made sense to match this format.

As normalization is not always necessary depending on the IDE, I decided to normalize the content in the IDE-specific part of Saros to avoid additional overhead created by unnecessary normalization. Even so, the utility to perform the actual normalization is still located in the Saros Core to avoid code duplication. For Saros/E and the Saros Server, I added logic to normalize all content before processing it for Saros purposes and then revert the normalization before applying content changes locally. For Saros/I an explicit normalization was not necessary as the IDE already normalizes the content to begin with. This is also explicitly mentioned in the code to avoid erroneous additions of the normalization logic in the future.

With these changes, I also introduced the implicit assumption that all content handled in the Core is normalized. As the normalization is handled by the IDE-specific logic and not the Saros Core itself, I extended the documentation of the corresponding Core components to explicitly state this assumption, requesting all passed content to be normalized. Furthermore, I added assertions checking for Windows line endings in central Core classes. This should make it easier to notice when an implementation does not normalize its content before passing it to the Core.

After introducing the assumption of working on normalized content in the Core, I simplified some of the logic added for the new positioning logic. Previously, the logic sometimes had to check which line ending was used in the given content to correctly calculate the position or the deltas produced by a specific change. This was done by looking for occurrences of Windows or UNIX line endings in the code. Such a check is now no longer necessary as the content handled by the Core now always uses UNIX line endings.

To finish the line ending normalization, I adjusted the consistency check in #878. This consistency check is run as part of the Saros watchdog component, ensuring that the shared workspace state is still synchronized for all participants. As the check uses the checksum and length of the shared content, it had to be adjusted to run on normalized content as well to avoid differing line endings being detected as an inconsistency.

[22] https://insights.stackoverflow.com/survey/2020#technology-developers-primary-operating-systems

With this final change, I had resolved all issues regarding the line ending handling described in section 9.1.

11 Preparing for the Filesystem Rework

After finishing my work on the line ending normalization implementation, I decided to work on resolving the issues with the current file system implementation described in section 9.2. The decision to resolve this roadblock next is discussed in section 12 covering the actual filesystem rework.

As the necessary refactoring of the filesystem handling was quite complex, I started by first cleaning up the surrounding logic, removing features that were made unnecessary with the filesystem rework, and simplifying the current filesystem interfaces. This made the actual refactoring of the filesystem logic substantially easier. Furthermore, working through the existing logic gave me a much better overview and understanding of the current filesystem implementation and its usage as well as its shortcomings. This made designing the new filesystem model easier.

11.1 Removing the Partial Sharing Logic

Saros supported a feature called "partial sharing". This feature allowed the user to select an arbitrary subset of resources from the current workspace to share instead of sharing a complete project/module. It was initially added to allow the user to select a minimal set of resources to share or to exclude specific resources from the session.

This feature was only supported by Saros/E and even there the implementation had multiple issues when used with anything but the most basic setups[23]. The general issue with this feature was that it allowed the user to choose *arbitrary* subsets of resources. As a result, it was possible to choose resources in a way that could not be represented in a single tree, i.e. by excluding a folder but including its child resources. Handling such arbitrary subsets is quite complicated as is shown by the existing faults in the logic.

With the rework of the filesystem implementation allowing the user to share arbitrary complete resource trees, the main feature of partial sharing — allowing the user to share a number of complete (sub-)trees of the workspace — is already available. This made the partial sharing feature almost completely obsolete. After discussing the issue with the rest of the Saros team, I decided to remove the feature. The removal of the partial sharing logic allowed me to simplify the existing resource sharing logic, making the following rework less complex.

To start the process of removing the partial sharing feature, I first just removed the UI capabilities to access it in pull request #880. This was done as, after the discussion on whether to remove partial sharing, it was decided that the feature should be removed before the Saros/E release `16.0.0` which was being prepared at that time. To not delay the release or add the need for further testing, the easiest and quickest option

[23] See issues #31, #36, #44, and #256.

was to only remove access to the feature instead of completely removing its logic from the code base.

I removed the complete partial sharing logic later on with the pull request #882. This allowed me to simplify the logic handling the resource negotiation and resource scope of a running session. Previously, the logic had to be able to handle specific sets of resources that were shared. Removing partial sharing meant that the handling could instead be restricted to only sharing entire projects/modules.

11.2 Reworking the Resource Transport Logic

To correctly synchronize the state of the shared workspace for all participants, Saros needs to be able to identify and access specific resources in the workspace. When only working in the local context, this can easily be accomplished by representing a resource as a composite of a project/module and a relative path.

This way of identifying resources cannot be directly applied when transferring actions to remote participants, as their local representation of the project/module might differ. Instead, Saros replaces the reference to the project/module with a unique ID representing it. The local mapping of the IDs is set up as part of the resource negotiation. This way of representing a resource in a client-agnostic way was done through the class `SPath`.

While this class was initially only meant to be used in the transport logic, over the years, it was also used in many other parts of the implementation of the Saros Core and the IDE-specific components to represent resources for non-transport-related logic. This made the general Saros filesystem concept and transport logic, as well as logic containing incorrect usages of `SPath`, harder to understand and maintain. Furthermore, the usage of `SPath` introduced issues that could have been avoided by instead using the Saros filesystem implementation directly. The process of resolving the resource described by an `SPath` object was not guaranteed to succeed, adding another point of failure that had to be handled. Furthermore, `SPath` did not provide a way of specifying the exact type of the resource that was referenced, leading to potential ambiguity if the referenced resource existed locally but was not of the expected type. Additionally, this ambiguity meant that `SPath` needed a way of resolving a local resource without knowing its type. As the Saros filesystem implementation does not offer a resource object of indeterminate type, the logic had to check for the type of the resource in the local workspace. Thus the conversion could only succeed if a matching resource existed in the workspace. This goes against the general design approach of the Saros filesystem implementation where all resource references are seen as handles for resources that do not have to necessarily exist in the workspace.

To resolve these issues, I decided to rework the resource transport logic. I started with pull request #934 by introducing the new class `ResourceTransportWrapper` to replace `SPath` in its intended usage. `ResourceTransportWrapper` uses generics to specify which resource type is contained. This allowed the logic to avoid the resource type ambiguity and removed unnecessary resource conversions. I also adjusted the classes using the `ResourceTransportWrapper` to only make the resource objects rep-

resented by the wrapper available to callers instead of the wrapper itself. This should hopefully avoid the transport class erroneously being used for non-transport-related logic.

Afterwards, with pull request #938, I replaced all other usages of `SPath` for unintended purposes, i.e. to identify resources outside of the transport context, with direct usages of the corresponding resource objects. This had been made much easier through the prior adjustment of the transport logic, as it now provides the correct resources objects directly. Subsequently, I deleted `SPath` as well as its related classes as its usage had been completely replaced. Furthermore, I adjusted the remaining references to the `SPath` concept to reflect the new logic.

11.3 Simplifying the Existing Filesystem Interfaces

As mentioned in section 3.1, the Saros filesystem implementation grew over the years, originating from the Eclipse filesystem interfaces. These interfaces were not adjusted during the extraction of the Saros Core logic. As a result, they still contained a lot of Eclipse-specific methods and method parameters. To make reworking the Saros filesystem implementation easier, I decided to first improve and simplify the existing Saros filesystem interfaces.

This process can be grouped into two categories:

- Removing no longer needed or unfitting classes, interfaces, and methods
- Simplifying existing methods and improving documentation

The following pull requests removed logic that was no longer needed or did not match the design of the filesystem logic:

#906 removed `IResource.move(...)`, a method used to move resources in the local filesystem. Moving files is a somewhat complex process that is handled differently by different IDEs. Creating an abstraction for this process complicated the implementation. As the method was never used in the Saros Core, such an abstraction seemed unnecessary. Delegating the logic to the IDE-specific parts of the implementation without the need for an abstraction instead allows for a tighter integration into the IDE filesystem logic.

#915 removed `IContainer.getDefaultCharset()`, a method that allowed requesting the default character encoding for a particular folder or project/module. The method was only used to request the default character encoding of the shared project/module for a compatibility check during the session negotiation. But, as the logic also does this for every shared file, it is very unlikely that the default project/module encoding is not already included in this check. Furthermore, this method assumes that every folder and project/module has a default character encoding. While this is true for Eclipse, this assumption does not necessarily hold for other IDEs. Implying such a granularity where it does not exist might lead to faults in the character encoding handling.

#916 removed `IResource.adaptTo(...)`, a method that could be used to safely trans-

form a resource object into a different resource type if possible. This method was a remnant of the initial implementation modeled after the Eclipse filesystem API. While such a method can be useful when the type conversion requires additional logic, it proved unnecessary for Saros, where it just checked if casting the object was possible. Wrapping this logic just made it harder to understand what the logic was doing and added unnecessary calls to the logic. Instead of wrapping this cast in the method, it is better to instead directly cast the object and only check for type compatibility if necessary.

#918 removed `FolderMoveActivity`, a class that could be used to signal the move of a folder to other participants. Packaging the move of a folder as a single activity had been a previous approach that was abandoned along the way. Instead, Saros now sends a creation activity for the new location of the folder and a deletion activity for the old location of the folder. This class was a remnant of the old approach that was never cleaned up.

#943 removed `IProject.findMember(...)`, a method that could be used to look up a resource object in the local workspace for a given path. The method was no longer being used after the removal of `SPath`. Furthermore, as explained in section 11.2, this method did not match the intended design of the filesystem. Instead of treating resource objects as handles, meaning the object can exist even if the resource it describes does not exist, the method could only ever return resource objects for existing resources.

#950 removed `IWorkspaceRoot`, the counterpart of an interface in the Eclipse filesystem API that describes the root resource of the local workspace. This concept does not match the design of the local workspace for other IDEs. For IntelliJ IDEA, for example, the workspace is a collection of modules, each operating on its own sets of paths. As a result, there is no single root that can be used to describe the entire workspace. Furthermore, the interface and its implementations were only ever used in the Eclipse-specific part of Saros, meaning it did not have to be available to the Core or other IDE implementations.

#953 consolidated the path handling in `IResource` by removing `getLocation()` and `getFullPath()`. In the previous setup, the Saros resource interface had three methods to access the path of a resource: `getLocation()`, providing the absolute path in the local filesystem; `getFullPath()`, providing the project name concatenated with the project-relative path; and `getProjectRelativePath()`, providing the project-relative path. These three methods were remnants of the Eclipse-based filesystem design and exactly match their Eclipse counterpart.

I removed `getLocation()` as it went against the design of the filesystem abstraction. The Saros filesystem interface is supposed to be an abstraction of the IDE-specific filesystem implementation that can be used by the Core. As such, it should have no need for absolute paths as they are not usable in the context of the Core. Logic that needs to operate on absolute paths most likely belongs into the IDE-specific parts of Saros, in which case it should use the IDE-specific filesystem API directly instead of relying on the Saros filesystem abstraction.

Still providing access to absolute paths led to hacks using them to replicate some IDE-specific behavior in the context of the Core as a workaround. To avoid this, restricting the access to absolute paths seemed sensible to me. For the one exception in the Core that still relied on absolute paths to uniquely identify resources in the local context — a cache for checksums of the content of shared files, used to speed up the resource negotiation — I added a separate utility class allowing the resolution of absolute paths for resources.

I removed `getFullPath()` as it worked based on the assumption that all resources are directly located below a project/module and that combining the project/module name and the project-/module-relative path of a resource will lead to a valid path for that resource. This is, however, not the case for other IDEs. In IntelliJ IDEA, a module just represents a collection of so called content roots. Each content root is a pointer into the local filesystem. In particular, this means that a module can contain a forest of resource trees. As a result, concatenating the module name and the module-relative path is not guaranteed to result in a unique path. Furthermore, the resulting path also does not have to be valid as the base folder of the content root does not have to have the same name as the module.

The usage of `getFullPath()` was replaced by using `getProjectRelativePath()`. This does not resolve the issue of module-relative paths not necessarily being unique, but this issue is related to much broader issues of the current filesystem implementation and the clash of the IDE-specific filesystem models that has been resolved with the rework of the Saros filesystem described in section 12.

#976 removed the `IWorkspace` methods `getLocation()` and `getProject()`. These methods were no longer being used. Furthermore, the assumption that the workspace can be represented by a single root resource does not hold for IntelliJ IDEA, making it impossible to implement the method in this case.

The following pull requests simplified the remaining filesystem interface methods and improved the documentation:

#907 removed unnecessary parameters from methods in different Saros filesystem interfaces and their implementations. These parameters were kept as a remnant of the initial, Eclipse-based design of the filesystem. As such, the parameters were highly Eclipse specific and therefore not supported by other IDEs. Furthermore, the parameters were not even used by the Eclipse implementation of the methods most of the time, meaning they were completely unnecessary. And, in the few cases where the parameters were actually being used, the given values were always the same. Such values were replaced by constants in the Eclipse-specific implementation of the method.

#908 introduced an enumerated type (i.e. enum) for the different resource types handled by Saros. Previously, as a remnant of the initial Eclipse-based implementation, these values were defined as static bit masks. As they were never used as bit masks in Saros, replacing them with an enum simplified the logic dealing with resource types.

#952 moved the methods `getFile(...)` and `getFolder(...)` from `IProject` to its parent interface `IContainer`. As the methods provide generic filesystem functionality that is not specific to projects/modules, they should be located in the more general interface to also apply for the other container implementations.

During this process, I also explicitly defined the expected behavior of the moved methods and adjusted the implementations to match this behavior. This was necessary as the behavior of the implementations for Eclipse and IntelliJ IDEA differed.

#977 added documentation to all filesystem interfaces and their methods, explicitly defining the provided functionality and expected behavior of the implementing classes. Having an IDE-independent documentation instead of just referring to the Eclipse API documentation is very important to create an interface that is actually implementable for different IDEs. Furthermore, having an explicitly defined expected behavior makes it a lot easier to create compatible filesystem implementations as part of the rework.

#978 added default implementations for methods with constant return values. This made the implementations of the interfaces cleaner by avoiding boilerplate logic that is always the same.

#1005 adjusted `IContainer.members()` to return a list instead of an array. This was done to simplify the current usage of the method which always creates a list from the array. Furthermore, the implementations already operated on a list internally, only converting it to an array for the return value. By directly returning the list, these two conversions can be avoided.

A UML diagram of the filesystem interfaces before and after the cleanup can be found in appendix B.1.

During these improvements of the Saros filesystem interfaces, I repeatedly encountered issues where the Saros Server implementation contained the last and only usage of a method that I wanted to remove. This was caused by the general design of the Saros Server filesystem implementation. While the implementations for Eclipse and IntelliJ IDEA both follow the project-/module-based model, meaning all resources are defined relative to a project/module, the implementation of the Server is centered around the workspace concept. All resources are defined in relation to the workspace. The so called "projects" in the context of the Server are just the direct child folders of the workspace root. As a result, the implementation heavily relies on the workspace concept, which was almost entirely removed as part of this cleanup. The problem with the Saros Server filesystem design is also described in the issue #980. Furthermore, as the Saros Server does not have a backing filesystem implementation, most of the Server-specific logic still uses the filesystem abstraction meant for the Core.

After discussing the issue with the rest of the Saros team, I decided to keep my work on the Saros Server filesystem implementation to a minimum. The Saros Server is still in an alpha state, has never been officially released, and did not have any developers actively working on it as of the time of writing. Investing a lot of time into adjusting

its filesystem implementation did not seem sensible to me. To keep the rework to a minimum, instead of reworking the Saros Server filesystem implementation, I kept the necessary methods that were removed from the interfaces around and accessed them directly instead of through the interfaces. This created additional technical debt in the Server implementation, but this was an acceptable tradeoff in my opinion.

12 Implementing an IDE-Independent Filesystem Model

The second roadblock I decided to address regarded the issues with the filesystem implementation described in section 9.2. This seemed to be the natural progression as resolving the roadblock seemed essential to providing IDE cross-compatibility. Before making this decision, I also looked into resolving the roadblock regarding the sharing scope described in section 9.3. I ultimately decided against working on it at that point in time as resolving the issue is not necessary for basic IDE cross-compatibility but rather improves the usability and user experience in cross-IDE setups. Furthermore, the logic deciding what not to share needs to integrate with the filesystem implementation, thus an improved filesystem logic would be beneficial. This is further discussed in section 14.1.

12.1 Solution Approach

The simplest solution approach would have been to keep the current filesystem implementation and exclude/forbid all cases that would lead to issues in cross-IDE sessions. While this approach would theoretically work, it would substantially limit the usable project/module setups, thereby severely reducing the usability of Saros. Furthermore, these restrictions would be very hard to convey to the user in an easily understandable fashion as they are the result of technical restrictions of the different IDEs. As a compromise, different sharing modes could be implemented for Saros, meaning that such restrictions would only apply for cross-IDE sessions. But this approach would only further complicate the existing sharing logic and introduce new complicated corner cases that would have to be dealt with.

In my opinion, using sharing logic that is tightly integrated with the IDE filesystem model is only an option when solely connecting to Saros implementations for the same IDE. Using concepts of the IDE filesystem model like projects or modules for the Saros sharing logic only functions well when both sides implement logic specific to that model. This is exactly what was done while implementing Saros/E and Saros/I in the past. The existing sharing logic heavily relies on assumptions of how the filesystem on the other side behaves and what model it uses. Otherwise, it is just too complicated to correctly map differing filesystem models in a way that allows for compatibility.

To avoid such issues, I suggest dropping the integration with the IDE filesystem model and instead make the sharing logic completely IDE-independent. As a consequence, this would also mean dropping the capabilities to create or adjust projects/modules as part of the Saros resource negotiation, requiring the user to do so instead. But this is a sacrifice that has to be made in my opinion. This comes down to what

functionality Saros aspires to provide. For me, it is providing developers with the means to collaboratively work on source code. Providing integration with the IDE workspace model is more the job of a build tool from my point of view. It falls outside of the scope of Saros' core capabilities and is too complicated to do well without requiring massive development efforts. And even then, it will be hard to cover even just the most basic setups, as the plugin would have to be able to work with other build tools (like Gradle or Maven[24]) which would usually provide the correct setup for the IDE-specific model.

To implement such an IDE-independent filesystem approach, I decided to follow the general concept described in the thesis by O. Hanßen (see section 5.4). The concept of sharing projects/modules is replaced by the concept of sharing so called "reference points". A reference point in this context is just a pointer into the local workspace. So, instead of choosing a project/module to share, the user would instead pick an arbitrary folder in the workspace to use as a reference point. All child resources of the chosen reference points would then be shared as part of the session. When joining a session, the user could simply choose which folder in the local workspace should represent which shared reference point. As each user has a mapping for the shared reference points, the path relative to the reference point for a resource can still be used to identify a specific resource for all clients.

In Hanßen's proposal, he also suggested separating the responsibilities of reference points. In the old filesystem model, the `IProject` class representing a project/module in the workspace had two different responsibilities:

- It provided access to all resources contained in the project/module represented by the specific `IProject` object. More specifically, it provided the capabilities of listing all resources contained in the project/module.
- It provided a reference point which could be used to create and resolve IDE-independent resource representations. More specifically, it provided a reference object with which specific relative paths could be created and resolved.

The proposed reference point concept instead should only provide the functionality to create and resolve IDE-independent resource representations. A result of this design choice would be that logic located in the Saros Core would no longer be able to request which resources are contained in a given reference point. As some of the Core logic still relies on these capabilities, such logic would have to be adjusted. To replace the split-off capabilities, Hanßen suggested adding an additional mapper that can convert between reference points and their resource representation.

I decided against following this design as I do not think that this split provides enough merit. While it makes the concept of a reference point cleaner, it would increase the effort necessary to adjust the existing logic to the new filesystem model. Not splitting these responsibilities allowed me to keep the existing filesystem interface and their usages. As a result, the Saros Core logic only needed very minor adjustments during the rework.

[24] http://maven.apache.org/

While designing the reference-point-based filesystem model, I encountered corner cases that I explicitly decided to exclude with this implementation. Nested reference points, meaning reference points that are contained in the resources of a different reference point, are not supported. In particular, this means that each reference point must represent a resource tree that is completely distinct from every other reference point resource tree. This restriction is necessary as nested reference points are very hard to handle. Two reference points that are nested in the local workspace could theoretically represent two completely distinct resource trees in a different workspace. This makes it very hard to correctly apply changes to such setups as some resources might belong to multiple reference points in some, but not all workspaces. This restriction is enforced by the Saros logic.

Not allowing nested reference points also makes it much harder to support expanding the sharing scope upwards. While the logic allows adding new reference points, expanding the scope by adding distinct resource trees, it does not allow expanding the scope by sharing parent resources of currently shared reference points. This logic would be necessary for the common use case scenario of first starting to work on a reduced set of files but then later deciding to expand the pair programming session to include a wider range of resources or even the entire workspace. Technically, this functionality could be added later on by dropping the reference points that would be included in the newly added reference point, but this falls outside of the scope of this thesis. Supporting this case would require a lot of complicated logic as all resources that were previously defined relative to the dropped reference points would have to be updated to be relative to the new parent reference point. As there is no central place handling and storing resource references, the logic would also have to ensure that the resource update is applied to every currently held resource reference. Furthermore, this would have to be applied retroactively to all activities in transit at that point, creating a possible race condition as it has to be made sure that no activities are processed locally before the reference point mapping has been completely updated.

For the rework of the filesystem implementations, I again kept the rework of the Saros Server to a minimum as it is still in the early alpha stages and is not in active development at the time of writing. As the Server filesystem implementation was IDE-independent to begin with, the existing logic should work relatively well with the other reference-point-based filesystem implementations. I have, however, not tested this, so this is only a guess on my part.

12.2 Reworking the Filesystem Implementation for Saros/I

I decided to start migrating the IntelliJ IDEA implementation of Saros first. Due to my work on Saros/I as part of my bachelor's thesis and my employment as a student researcher, I am very familiar with the existing logic, the IntelliJ IDEA APIs that were used, and the IntelliJ IDEA filesystem model. Starting with Saros/I allowed me to gain experience and discover issues to look out for while still in a familiar context, making the following reworks easier.

To start, I introduced the new reference-point-based filesystem implementation as well as a utility class to convert between the new implementation and the IntelliJ

IDEA filesystem representation with pull request #997. As the previous filesystem design already defined all resources relative to `IProject`, i.e. the Saros wrapper of a module, I could model the new implementation after it by simply replacing the concept of wrapping a module with wrapping the resource representing a reference point. This also allowed me to reuse parts of the existing code, further reducing the effort required to create the new implementation. Afterwards, I proceeded migrating the different components of the IntelliJ IDEA implementation to the new filesystem implementation.

I started with migrating most of the internal components with pull request #998, mainly replacing usages of the old filesystem implementation. Additionally, I removed some module-specific logic that was no longer necessary with the new filesystem model. This process continued with pull request #1001, adjusting the logic handling local and remote resource changes. I removed the logic handling specific corner cases related to modules and adjusted some of the special case logic to apply to reference points instead.

Next, I migrated the UI logic to support the reference-point-based sharing model with pull request #999. This proved easier than expected as the feature set of Saros/I is still restricted. It only allows sharing a single reference point and does not allow adding new reference points during the session. As a result, most of the UI elements could remain unchanged. Furthermore, no longer supporting the creation or adjustment of modules as part of the negotiation allowed me to drop a lot of the more complicated UI-related logic. The wording used in the UI was also adjusted with pull request #1008.

Lastly, I cleaned up the remnants of the old filesystem logic with pull request #1000. This included removing the old implementation and updating all references to it or the project-based filesystem model in documentation as well as class, method, and variable names.

As the IntelliJ IDEA implementation of Saros is not covered by the Saros Testing Framework, I instead had to rely on manual testing to ensure that the rework did not break any functionality. For these tests, I proceeded just like when preparing for a release, testing all available features, paying special attention to potential issues with the consistency of the shared workspaces.

12.3 Cleaning up the Saros Core

After finishing the migration of Saros/I, I continued with migrating the Saros Core. As the filesystem interfaces remained unchanged, I had to adjust very little of the actual Core logic. Instead, with pull request #1012, I cleaned up the Core by replacing references to the old filesystem implementation in documentation as well as class, method, and variable names.

The only exception was a small part of the sharing logic that contained a specific handling for sharing Eclipse project configuration files. As the logic was no longer needed and did not belong in the Core to begin with, I removed it.

12.4 Reworking the Filesystem Implementation for Saros/E

Migrating the Eclipse implementation proved to be a lot more tricky than the IntelliJ IDEA implementation. This was partly due to me not being as familiar with the implementation of the Eclipse filesystem model and partly due to Saros/E being in development for longer. It provides a much wider range of features which have grown over the years through the work of many different developers. As a result, it carries technical debt caused by workarounds introduced to accommodate the rapid feature growth. Furthermore, to provide better usability, many of the additional features provided a better integration with the IDE, specifically the workspace model. While this was most likely the right decision at that time to provide the best user experience possible, it also made migrating the implementation to a more IDE-independent filesystem model much more difficult.

I again started the migration by introducing a new, reference-point-based filesystem implementation with pull request #1023. This required more effort than the previous migration of the IntelliJ IDEA implementation as the existing filesystem logic had to be completely rewritten. The previous implementation was largely a wrapper for the Eclipse filesystem API. As such, it matched the resource hierarchy structure provided by Eclipse. This was not an issue with the project-based sharing model as the Eclipse filesystem model also defined all its resources relative to the project they belonged to. But, with this new sharing model, resources are defined relative to an arbitrary folder in the workspace. This cannot be easily represented by solely relying on Eclipse resource objects. The difference in abstraction level between the previous implementations for IntelliJ IDEA and Eclipse made a big difference in this case. While the previous implementation for IntelliJ IDEA defined all its resources relative to the Saros wrapper of a project/module, the implementation for Eclipse defined them relative to the Eclipse representation of a project/module itself. As a result, for the IntelliJ IDEA implementation, I was able to just replace the concept of wrapping a project/module with the concept of wrapping a reference point. In the Eclipse implementation, on the other hand, I had to completely redesign and rewrite the logic to define resources relative to the Saros wrapper for a reference point instead of relying on Eclipse resource abstraction. I largely based the new design on the one used for the IntelliJ IDEA migration. As an added benefit, this also resulted in the implementations for Eclipse and IntelliJ IDEA being similar, making them easier to maintain in the future. A utility class to convert between a Saros resource object and the corresponding Eclipse filesystem representation was also added to make the usage and integration of the new filesystem implementation easier.

I started adjusting the existing logic to use the new filesystem implementation by first migrating various internal classes with #1032. These changes mostly boiled down to replacing the usage of the old filesystem implementation with the new one. Subsequently, I migrated the logic handling local changes in the filesystem with #1033. As part of this migration, I also optimized the logic to only check for changes of shared resources instead of looking at all changed resources and then checking for each of them whether it is actually shared. This should reduce overhead, especially for the handling of bulk operations like refactorings affecting multiple files.

After migrating the general internals, I moved on to migrating the UI. This required considerably more effort than the rest of the migration process. I started by migrating the UI internals with pull requests #1034, #1036, and #1039. Again, most of the changes were just replacing the usage of the old filesystem implementation with the new one. But I also spent some time introducing some small optimizations. With the new filesystem implementation, the conversion of Eclipse resource objects to Saros resource objects got significantly more expensive. While the old logic could run this conversion in $\mathcal{O}(1)$ time as the old filesystem implementation was just a wrapper for Eclipse filesystem objects, the new conversion logic requires $\mathcal{O}(n)$ time where n is the number of shared reference points. This is due to the conversion logic now having to check which, if any, of the shared reference points the resource belongs to. Most of the time, this is not an issue as the number of shared reference points is generally quite low. Furthermore, such conversions are usually only done for a small number of resources at a time. But there are specific cases in which a large number of resources is converted at once where this increase in runtime complexity might have a negative impact on performance. An example of such a case is adding shared resources to the session, e.g. during the session start. During a session, all shared resources are marked with a special icon in the project view to indicate that they are shared through Saros. When shared resources are added, these icons have to be updated. To accomplish this, the logic iterates through all resources in the workspace. For each resource, it has to apply the resource conversion as it needs the Saros resource representation to check whether it is shared. Even though this process happens in a background thread, it could still put substantial strain on the system performance. To alleviate this issue, I introduced a caching mechanism holding the last used reference point in the corresponding UI logic. This reduces the conversion time to $\mathcal{O}(1)$ in most cases as the resource trees are processed in depth-first order, meaning all resources of a reference point will be processed before proceeding to the next reference point. Therefore, an actual reference point lookup requiring $\mathcal{O}(n)$ time is only necessary for the first resource processed for each reference point.

Next, I migrated the wizard used when starting a session or adding resources to a running session with #1035. The wizard allows the user to select the resources to share from a checkbox-tree showing the resources in the workspace. The migration of the wizard proved difficult for multiple reasons. Over the years, a lot of additional features, most notably saving and restoring specific selections as well as undoing and redoing changes, had been added to the wizard. A lot of these features were not integrated very well into the core selection logic, meaning they had to be rewritten as part of the migration. Furthermore, as the handling of the selected resources was still a remnant of the partial sharing feature, it had to be rewritten as well. The most pressing issue, however, was the checkbox-tree implementation provided by the Eclipse API, as it did not offer some of the needed functionalities. With the new reference-point-based sharing model, users can only share complete resource trees. To match this restriction, they should not be able to deselect a resource located below a selected reference point. Additionally, they should not be able to select resources in a way that would lead to the creation of nested reference points. These restrictions should be enforced by the wizard, not allowing the user to make illegal changes.

But the used checkbox-tree implementation does not provide a way of disabling tree elements, making it very hard to enforce these restrictions. As I did not want to write a complete new checkbox-tree implementation, I decided to instead revert illegal changes made by the user. This happens transparently to the user, meaning to them it just looks like the change never happened. While this does not provide the best usability, I did not see any other way around this issue in the given time frame. This issue is also expanded upon in section 14.2.

As the last UI component, I migrated the wizard to select how to map the shared reference points in the local workspace when joining a session with #1037. The same issues as before also applied here. The wizard contained a lot of logic specific to the previous resource model, meaning it had to be rewritten almost entirely. Furthermore, during the rework, I discovered a general problem with the Eclipse workspace model. For a resource to be recognized by the workspace, it has to be loaded as part of a project. Even if the resource is present in the filesystem, it won't be accessible in the workspace unless it is part of a project. As a result, the wizard still has to support creating a project and using it to represent a shared reference point to cover cases where there are no projects in the workspace. While it would have also been possible to halt the negotiation and request the user create a project before continuing, this would have provided poor usability. Even though this technically provides the option to create a project as part of the resource negotiation, which is what I wanted to avoid with the new sharing model, it is in no ways comparable to the previous project integration which aimed to correctly set up the shared project in the local workspace. Instead, the new logic just creates the most basic project setup possible to meet the minimum requirements to map resources into the workspace. The actual configuration of the project still has to be done by the user after the negotiation if they want to use IDE features like specific language support. As a side effect of this new handling, when creating a new project to represent a project completely shared through Saros, the project configuration that is transferred as part of the resource synchronization will be automatically picked up by Eclipse. In such cases, no further configuration by the user would be necessary, providing almost the same usability as fully supporting setting up the project.

After migrating the UI, I proceeded with cleaning up the Eclipse implementation with pull requests #1038 and #1049. I removed the old resource implementations and replaced all references to the old sharing model in documentation as well as class, method, and variable names, completing the migration of the Saros/E source code. With these changes, I had resolved all issues regarding the filesystem implementation described in section 9.2.

12.5 Adjusting the STF after the Saros/E Migration

To complete the migration process to the new filesystem model for Saros/E and ensure that it did not introduce any new faults, I adjusted the Saros Testing Framework (STF) to work with the changed UI components with pull request #1040. As the STF identifies UI components by their names and displayed content, I had to adjust the test logic to match the new wording. Furthermore, I had to adjust the interaction with

UI components whose fields had changed as part of the rework.

While this migration process seemed quite simple in the beginning, with the tests passing locally after only some minor adjustments, getting them to run successfully as part of the Saros project continuous integration (CI) setup presented more of an issue. When running the test suite through Docker[25] containers as part of the CI setup, multiple tests failed with a timeout while trying to open an editor for a shared resource for some of the non-host participants. This was due to the project no longer being correctly registered as a Java project for some of the invited participants. While the project configuration was correctly set up as part of the resource negotiation with the previous sharing model, the new sharing model only creates a very basic project without any language support. Some of the STF APIs used to access the shared files through the Eclipse UI differentiate between files and classes. As a result, when the test looks for a class in a project that is set up without Java language support, it won't find the file, even if it exists in the workspace. I resolved this issue by adjusting the logic used to set up the test sessions in cases where the projects require Java language support. Whenever a new project should be created as part of the test setup, instead of solely relying on the Saros resource negotiation to create the project on the invitee's side, a stub project with the correct language support is created on the invitee's side which is then used for the resource negotiation. This approach mimics the behavior of creating a new project as part of the session setup while still guaranteeing that the project is configured correctly on the invitee's side. To make using this new setup process as easy as possible, I added utility methods that can be used to set up a session with Java language support. Furthermore, I adjusted the documentation of the STF methods allowing test setups that create a project as part of the resource negotiation to mention that they should only be used in cases where the project language support is not of interest.

Analyzing and resolving this issue required a lot of work as the testing conditions were quite unfavorable:

Firstly, as the problem lay with the STF API, neither the log files nor the screenshots of the failure state provided by the CI environment showed any obvious issues. Even though the screenshots showed that the project did not support any Java language features, this was quite hard to notice without explicitly looking for it. The fact that the project did not provide Java language support was only indicated by the absence of a small "J" above the project icon.

Secondly, every test run took around 40 minutes. This meant that it was very cumbersome running the tests multiple times to analyze the issues. Running the tests separately was not an option as some of the test failures could not be observed in such a setup. Furthermore, the test order was not predetermined. So, even though the desktop environment of the Docker container for the test workers could be accessed through VNC[26], it was difficult to figure out when to actually observe the test workers without having to sit through the entire test run.

[25] https://www.docker.com/
[26] https://en.wikipedia.org/wiki/Virtual_Network_Computing

Thirdly, the test failures were very inconsistent. The set of failed tests often changed by quite a lot, making it hard to analyze the issue across subsequent test runs. This was due to intricacies with how Eclipse handles resource refreshes. As part of the session setup, the `.project` file defining the language support was transferred to the invitee's workspace. Reloading this file would result in the project correctly providing Java language support, thereby fixing the test issues. While Eclipse should automatically pick up on such changes to the `.project` file, this does not always seem to happen immediately. Rather, from what I can tell, Eclipse seems to periodically check and refresh these files if necessary. As a result, some of the tests would succeed if, by happenstance, the `.project` file was reloaded at the correct time. This also explains why the tests repeatably ran with a higher success rate when executing them separately in the docker container. As the tests were run directly after the container was set up in such cases, the timing was always the same, making it more likely that a successful test run was reproducible.

Fourthly and finally, I was not able to properly reproduce the issues locally. All tests passed when running them directly through local Eclipse instances. This was due to Eclipse always directly picking up the changed `.project` file, meaning the Java language support was always loaded in time. I am not quite sure as to why this was the case when running the test locally, but from what I can tell, the reload seems to get triggered directly through user interactions. As the tests run in an encapsulated environment for the CI setup, there never is any user interaction to trigger such a direct reload. This would also explain why the CI tests ran with a higher success rate whenever I was observing them directly through VCN. Observing the test workspace again created the possibility of user interaction triggering a `.project` file reload, thereby allowing the test to run correctly.

As resolving this issue required far more time than I initially expected, I subsequently did not have the time to extend the existing STF tests to cover the new cases introduced through the reference-point-based sharing model. The new cases to cover are described in issue #1045.

12.6 Preventing Nested Reference Points

After completing the migration to the reference-point-based filesystem model, I also decided to add a check to the Saros Core ensuring that no nested reference points are shared. This check was added with pull request #1058.

Even though this check should not strictly be necessary as nested reference points should be prevented by the UI of the different Saros implementations, it is still nice to have a sanity check in the Core. This ensures that, even if a Saros implementation handles reference points incorrectly, the faulty behavior will still be caught, making it easier to discover and debug the issue.

As the Core does not have the capabilities needed to compare two reference points as it can't access the underlying resource represented by a reference point, this functionality had to be provided through the `IReferencePoint` interface. For this purpose, I added the method `IReferencePoint.isNested(IReferencePoint)`, checking whether

the given reference point would create a nested configuration with the called-on reference point. The method is used in `SharedReferencePointMapper`, the central class managing the shared reference points, checking that the addition of a new reference point would not create nested reference points.

12.7 Moving the Checksum Cache into the Session Context

During the resource negotiation, Saros creates a checksum for every shared file on the sender and receiver side to easily compare the file content. As this process is somewhat computationally intensive, a checksum cache was implemented to speed it up by reusing previously computed checksums. This substantially speeds up parallel or subsequent negotiations.

As this logic is located in the Saros Core, it relies on the Saros filesystem implementation to identify the different cached resources. This became a problem with the new reference-point-based filesystem model, as resource objects are now defined relative to reference points. Since reference points only exist during a session, resource objects can only be created during a session. Furthermore, for subsequent sessions, different reference points could be chosen, making the existing resource representations unusable even if some of the same resources were shared again. As a result, it was no longer possible to maintain the cache between sessions, meaning it could not be correctly adjusted for local changes in such cases.

I found two possible solutions for this issue:

- Making the caching logic generic, allowing the usage of IDE-specific resource objects instead of Saros resource objects.
- Moving the caching logic into the session scope, meaning it only exists during a running session.

After a long discussion with other Saros team members, I decided against making the caching logic generic. While this would enable the checksum cache to still cover all use cases, it would also make the existing logic more complicated and harder to use. Furthermore, I don't think that the gained speedup for subsequent session starts is as important. In my opinion, the main use cases are parallel resource negotiations and additional resource negotiations during a session. The case of parallel resource negotiations is important, as separately computing the checksums for all shared files for each negotiation could put substantial strain on the system, especially with larger session scopes and/or a higher number of session participants. The case of additional resource negotiations during a running session is important, since Saros is already quite resource intensive. Adding additional load might slow down the system. This could disrupt the workflow of the user or at least lead to a degraded user experience. The case of subsequent session starts, on the other hand, does not gain as much from the speedup in my opinion. As the user has to manually start or join a new session, their normal workflow is halted anyway, meaning the additional time needed to repopulate the checksum cache should not matter that much. Additionally, I did not like the idea of Saros using up system resources without providing any actual benefit or functionality by having a component running in the background even though there

is no active session.

Instead, I decided to move the caching logic into the session scope, adjusting the logic with pull request #1072. This preserved the main benefits of the cache, speeding up parallel resource negotiations and additional resource negotiations during a session, while minimizing the overhead created while there is no running session. In addition to resolving the checksum cache issue, moving the cache to the session scope also allowed me to clean up the logic surrounding it. This simplified the setup for the checksum cache for Eclipse and removed a long-standing workaround introduced to avoid circular dependency issues related to the cache.

13 Other Resolved Issues

Besides resolving the identified roadblocks, I also resolved multiple smaller issues related to IDE cross-compatibility and the general reliability of Saros as part of this thesis.

13.1 Implementing a Unified Handling of Non-text Editors

The main purpose of Saros is to allow developers to collaboratively work on shared documents. This still leaves the question of how non-text editors should be handled. This handling did not match for Saros/E and Saros/I. While Saros/E completely ignored non-text editors, Saros/I tried to handle them like normal editors, leading to some issues.

To unify the behavior, I chose ignoring non-text editors as the default behavior for Saros, adjusting the IntelliJ IDEA implementation to match it with #842. This somewhat reduces the usability of the follow mode as it now no longer applies to non-text editors like images. But, while it might be useful for such resources to be visible for other participants during pair programming, this is a rather limited use case. Furthermore, if this use case were to present itself as important in the future, the logic of both implementations could still be adjusted. Creating a unified behavior to allow for IDE cross-compatibility was the more important goal for now.

13.2 Fixing an Undiscovered Fault in the Operational Transformation Logic

During my work implementing line ending normalization, I discovered a fault in the operational transformation logic that had gone unnoticed for about 12 years. An optimization to combine two insertions that can be expressed as a single insertion operation was implemented incorrectly, combining actions that cannot be correctly represented by a single insertion. The same error that had been made implementing this optimization was also made when implementing the corresponding tests, meaning the fault remained undiscovered.

The logic incorrectly tried to combine insertions where the start of the first insertion equals the end of the second insertion by simply appending the first insertion to the second one. But since the second insertion shifts everything after it to the right, this

case leads to a situation where the two insertions are separated by preexisting content, meaning they cannot be represented by a single insertion operations.

The correct handling is to check that the start of the first insertion is the same as the start of the second insertion. This ensures that the insertions are not separated by any other characters, meaning they can be combined into a single insertion operation. This combined insertion has the same start as the second insertion and contains the content of the first insertion appended to the end of the content of the second insertion.

An example of both of these cases is shown in figure 13.1. It shows first inserting the text "cd" and then the text "ab" into the initial text content "01234567", (a) depicting the case that should be combinable according to the old logic and (b) depicting the case that should be combinable according to the new logic. This example makes it very clear that the insertions covered by the old logic can't be combined. As the insertion of the content "ab" shifts the content currently at that position to the right instead of replacing it, the insertions of "ab" and "cd" are separated by the initial content "12". Such a non-continuous insertion cannot be represented by a single insertion operation. The case covered by the new logic, on the other hand, leads to a continuous insertion that can be represented by a single insertion operation. The corresponding combined representation would be inserting "abcd" at position 3.

	0	1	2	3	4	5	6	7
Initial	0	1	2	3	4	5	6	7
Ins1	0	1	2	c	d	3	4	5
Ins2	0	a	b	1	2	c	d	3

(a) Insertions that can't be combined.
Ins1 inserts "cd" at position 3.
Ins2 inserts "ab" at position 1.

	0	1	2	3	4	5	6	7
Initial	0	1	2	3	4	5	6	7
Ins1	0	1	2	c	d	3	4	5
Ins2	0	1	2	a	b	c	d	3

(b) Insertions that can be combined.
Ins1 inserts "cd" at position 3.
Ins2 inserts "ab" at position 3.

Figure 13.1: Cases covered by the (a) old and (b) new optimization logic.

I corrected the logic and adjusted the corresponding test to reflect the corrected behavior with pull request #868.

13.3 Implementing a Unified Handling for File Moves

Eclipse and IntelliJ IDEA handle the move of files differently. In IntelliJ IDEA, the move of a resource only consists of changing its path. But in Eclipse, it may also contain a change of its content in addition to changing its path. As a result, resource move activities created by Saros for Eclipse may also transfer the adjusted content for the moved resource.

This adjusted content was not honored by the Saros/I implementation, instead always only moving the resource. With #905, I changed the logic to also adjust the content accordingly if the received move activity transfers the adjusted file content.

13.4 Implementing a Unified Handling for File Character Encodings

When sending an activity representing the creation or move of a file, Saros also provides the option of sending the binary content of the file. Along with this binary content, the character encoding used for the content can be sent as well. This character encoding parameter was optional. While Saros/I and the Saros Server always included the used encoding, Saros/E did not. Instead of explicitly sending the character encoding for every file, it instead relied on the character encoding configuration that was set up along with the project configuration at the session start. Since adjusting the project configuration as part of the session start is now no longer supported, this behavior is no longer an option for Saros/E. Instead, it should also rely on the character encoding passed as part of the corresponding activities.

I unified the character encoding handling with #913. With it, I introduced a central method to set the character encoding for a file as part of the Saros file interface. Subsequently, I adjusted all Saros implementations to always use the received character encoding when applying received activities containing binary content. Furthermore, I adjusted the Saros Server implementation to use the held character encoding when reading content from disk. Previously, the logic relied on the Java virtual machine default encoding in such cases, which only worked in cases where the default encoding matched the encoding used by the other participants.

Additionally, I added a check requiring activities containing binary content to also contain the character encoding to use with #914. This ensures that the encoding is always available when necessary. Subsequently, I also adjusted the Eclipse implementation to comply with these restrictions, always sending the used character encoding alongside binary content.

13.5 Introducing Improved Local Resource Mapping Suggestions

With the previous, project-based sharing model, an incoming resource negotiation asked the user which local projects/modules they would like to use to represent the shared remote projects/modules. As project/module names were usually unique in the workspace and easily recognizable, it was relatively easy for the user to determine the correct local mapping for the shared resources.

With the new, reference-point-based sharing mode, however, the incoming resource negotiation asks the user which local folders they would like to use to represent the shared remote folders. As these folders can be chosen arbitrarily, the names in no way have to be unique or meaningful in the context of the local workspace, meaning the user often has no real indication which local folders to use for a correct mapping. A worst case example for this issue would be sharing a folder with an ambiguous name like “src”. Such a folder can easily exist multiple times in the local workspace, providing no hint which source folder was actually shared by the sender of the negotiation. This is a big shortcoming in the new sharing logic.

To somewhat alleviate this issue, I introduced logic to provide a better default suggestion on how to map each shared resource in the local workspace with pull request #1051. To do so, the logic calculates the relative path between the shared reference

point resource and each relevant reference object in the local workspace. For Eclipse, such relevant reference objects would be the loaded projects. For IntelliJ IDEA, such relevant reference objects would be the modules in which the reference point is contained. As IntelliJ IDEA supports nested modules, it is important to include the relative path to all parent modules and not just of the closest one. With certain setups, the equivalent to the Eclipse project for a resource might not be the closest module but one of its parent modules. An example of this is an IntelliJ IDEA module set up through Gradle, which creates dedicated nested modules for the source and the test folder.

These possible mappings are then transferred as part of the resource negotiation data, meaning the receiving side can use them as suggested default mappings. As the names of important reference objects such as projects or modules often match across IDEs, this allows for an accurate default mapping, even if the name of the reference point resource is ambiguous, by resolving the given relative path against the reference object in the local workspace.

IV Conclusion

This part presents the conclusion of the thesis. It starts with section 14 giving an overview of open topics and issues left unresolved as part of this thesis. Afterwards, section 15 gives an evaluation of the thesis, its process, and its results. Lastly, section 16 proposes possible next steps as well as long-term goals building on this thesis.

14 Open Topics and Issues

At the end of the thesis, there still were some open topics and unresolved issues related to IDE cross-compatibility. These topics and issues are discussed in this section.

14.1 Implementing a New System to Determine What Not to Share

As part of this thesis, I did not have the time to resolve the roadblock concerning the resource exclusion logic described in section 9.3. While I started implementing a prototype for a possible solution, I quickly decided to switch to reworking the filesystem handling instead. As the scope of the thesis along with the necessary effort kept increasing, it became clear to me that resolving all roadblocks had become unlikely. Since resolving this roadblock was not essential to providing IDE cross-compatibility but instead just increases the usability of Saros in cross-IDE sessions, I decided to shift my focus to more important issues. Furthermore, while designing the prototype, it became apparent that the exclusion logic would also benefit from an improved filesystem implementation.

Nevertheless, I still proposed and discussed different solution approaches for the roadblock. The general issue of defining the exact session scope can be split into two parts:

1. How should Saros determine which resources should be excluded from the session scope?
2. How should Saros ensure that the set of ignored/excluded resources matches for all participants?

The simplest solution approach for both issues would be to avoid restricting the session scope at all, forcing the user to always share all files contained in the chosen reference point. But, as mentioned in the description of the roadblock, there are good reasons not to share particular files, e.g. because they are build artifacts or contain user-specific data.

Regarding the question of how Saros should determine the set of ignored resources, keeping the current implementation would be an option. This would avoid having to implement a new system. But, as a drawback, it would force Saros to completely

rely on the IDE-specific concepts to ignore resources. Furthermore, it would force users to add files to the IDE-specific model to ignore resources only for the purpose of Saros, potentially inhibiting IDE functionality. With IntelliJ IDEA, for example, excluded resources are not indexed by the IDE, meaning they can't be found through the different workspace searches and are ignored by the IDE refactoring tools.

An alternative to relying on the IDE-specific logic determining what is not supposed to be shared would be implementing a completely independent system for this purpose instead. This system could be designed using a general interface that can accept multiple replaceable implementations, making it possible to offer different ways of specifying which resources to ignore. A simple approach would be offering a dedicated `.sarosignore` file modeled after the git-ignore logic, specifying which resources to ignore. Prime candidates for additional implementations would be integrations with version control systems as they already provide information on what not to keep under versioning, i.e. what not to share in the context of versioning. Most of the time, the files not under versioning will align with the files that should be excluded from a Saros session. As Git is currently probably the most prevalent version control system, it would make sense to start by offering a Git integration.

Regarding the question of how to ensure that all participants have the same set of ignored resources, there are two approaches: either negotiate a shared set of ignored resources between all participants, or have the session host dictate which resources are ignored.

Negotiating the shared set of ignored resources would offer the best usability, as all local user preferences are honored. But it also would make the already-complicated Saros negotiation logic even more complicated since it would have to rely on a separate negotiation taking place before the shared workspaces are synchronized. Furthermore, this approach would require the adjustment of the shared set of ignored resources whenever a new participant joins the session. This would create the possibility of reducing the shared scope during a session, which is generally not supported by Saros.

Having the set of ignored resources dictated by the host would somewhat reduce the usability, as the local preferences would no longer be honored for the other participants, but this issue could be avoided by the participants reaching an agreement on which resources to ignore before the session. While this approach would still require the transfer of the set of ignored resources, the set would be determined by the host, meaning no negotiation would be needed. As a result, this additional information could simply be piggybacked onto existing resource negotiation data. Furthermore, having the host dictate which resources are ignored makes it much easier for the users to determine the exact scope of the session. When honoring the exclusion options of all participants, the resulting session scope might be unexpected for some participants as there might be different preferences/opinions in this regard.

Integrating this new resource exclusion system into the existing Saros logic should be relatively easy by just replacing the existing calls to `IResource.isIgnored()`. This method is the central way of determining which resources to exclude from the ses-

sion scope in the current implementation, so just replacing its usage should cover all necessary logic.

14.2 Eclipse UI Improvements following the Filesystem Rework

The rework of the Eclipse filesystem implementation described in section 12.4 led to usability issues for both the wizard used to send and the wizard used to accept resource negotiations.

For the wizard used to send resource negotiations, i.e. allowing the user to choose which resources to share as part of a Saros session, the checkbox-tree component used to display the resources in the workspace needs to be replaced. The current implementation provided by the Eclipse API does not allow disabling tree elements. As a result, I had to introduce many workarounds to prevent the user from selecting invalid resource setups. These workarounds led to the tree behaving in somewhat unexpected ways while also providing very poor user feedback. A better solution would be replacing the checkbox-tree implementation with one that allows disabling elements. As far as I can tell, no such implementation is provided by the Eclipse API, so it would have to be implemented from scratch. Since the component is tightly integrated into the Eclipse UI logic, creating new implementations might prove difficult. This issue and the given solution approach is described in more detail in issue #1041.

An additional issue with the current checkbox-tree implementation is that it does not function correctly with older Eclipse versions. In cases where resources are preselected by default when the wizard starts, the tree should auto-expand to show all selected resources. While this is implemented through the appropriate API, it does not seem to function with older Eclipse versions, so it is most likely an upstream issue with the Eclipse API. This issue is described in more detail in issue #1055.

For the wizard used to accept incoming resource negotiations, i.e. allowing the user to choose how to represent the shared resources in the local workspace, a better way of differentiating between the shared reference points has to be provided. The wizard contains a separate tab for each reference point that is part of the negotiation. Those tabs are identified through the name of the shared reference point root on the sender side. As already stated before, the names of reference point roots don't have to be unique in the context of the workspace or even the resource negotiation. This could make it very difficult differentiating the different tabs and choosing an appropriate local representation for each shared reference point. While this issue has already been somewhat alleviated through the introduction of the improved default mapping suggestions described in section 13.5, it should still be properly addressed as the proposed mappings might not always be usable. My proposal to provide better usability would be to also display the workspace-relative path of the reference point root on the sender side. This would help the local user to identify the correct reference point resource in the local workspace. Additionally, displaying a preview of the resource tree represented by the reference point on the sender side would also be helpful. This would give the user a better understanding of what will actually be shared as part of the Saros session. This issue and the suggested solution approach is described in more detail in issue #1042.

An additional issue with the wizard is that it does not scale correctly for some high-DPI setups. High-DPI setups use displays with a high pixel density (i.e. a high resolution paired with a small screen size), requiring the OS to scale up the displayed components to make them legible. With such setups, the height of the wizard is not set correctly, leading to the trailing options not being visible with the default size. The wizard also does not display any indicators that there is content that is not currently visible. As a result, a user could potentially be left unaware of some of the options to represent a reference point in the local workspace. Since every option represents a common use case, this would severely reduce the usability of the wizard. Most likely, this is caused by an upstream issue with the Eclipse API used to size the wizard. This is described in more detail in issue #1043.

14.3 Extending the Existing Character Encoding Handling

Even though the character encoding handling has already been improved as part of this thesis (as described in section 13.4), it is still incomplete.

With the current implementation, only the character encoding for binary content that is transferred during a session, i.e. when creating a new shared file, is set. When transferring binary content during the resource negotiation, i.e. to create files on the receiving side to synchronize the initial workspace state, the character encoding is not included. Instead, the encoding configuration of the local workspace is used. Since Saros no longer adjusts the workspace settings as part of the session setup, it is not guaranteed that the character encoding used as the workspace default is usable for the sent content. To avoid the issue, the character encoding for each transferred file should be included in the negotiation. This ensures that the content of the files created as part of the negotiation will be read correctly by the IDE. This problem as well as the suggested solution approach is described in more detail in issue #912.

While this would cover the last case where Saros transfers new content, it does not cover changes to the character encoding of existing files. In most IDEs, the user has the option to change the character encoding of a file. Most often, this results in the current content of the editor for the file to be re-written to disk using the new character encoding. Such changes are currently not explicitly supported by Saros. There is no specific activity that is documented to handle this case. While the Eclipse implementation uses the file creation activity to cover this case, it would be nicer to have a separate activity type for it. Or, even if the usage of the file creation activity remains for this case, it should at least be included in the activity documentation. This problem is described in the issues #940 and #941.

14.4 Introducing a Unified BOM Handling

For some Unicode encodings, a byte order marker (BOM) can be included at the start of the file to provide meta information like the used endianness. When accessing the binary content of a file on disk through the IntelliJ IDEA API, the BOM is skipped. It has to be requested separately through the usage of a specific method. For Eclipse, on the other hand, the BOM is included in the normal binary content. This problem

is discussed as part of issue #707.

To avoid potentially dropping the BOM, a unified handling should be implemented for Saros. It has to be decided whether the BOM should be included in the binary content by default or whether it should be sent as a separate value. Subsequently, the filesystem implementations need to be adjusted accordingly.

15 Evaluation

The main goal of this thesis was to provide IDE cross-compatibility for Saros, allowing users to create Saros sessions between Eclipse and IntelliJ IDEA instances. As part of this thesis, I identified key roadblocks regarding supporting cross-compatibility and proposed solution approaches for them. Furthermore, I resolved the essential roadblocks and other issues regarding base Saros functionality, enabling basic Saros IDE cross-compatibility. All created patches were peer-reviewed by other members of the Saros project and successfully merged back into the central codebase. While there are still open issues regarding cross-compatibility not resolved by this thesis due to time restrictions, I assert that the overall goal has been reached. With the implemented changes, Saros can create sessions between instances running on Eclipse and IntelliJ IDEA, allowing the user to work collaboratively on all shared resources. The remaining roadblock and issues mainly relate to improvements in the usability of cross-IDE sessions, covering a wider range of features, and resolving outstanding bugs. Additionally, I discovered and documented possible solution approaches for issues not resolved as part of this thesis, making it easier to resolve them in the future.

The UI rework done as part of the introduction of the reference-point-based sharing model was also restricted by time. Even though I expected that the change of such a central concept would require UI adjustments, I underestimated the extent to which the UI had to be adjusted or completely rewritten. As a result, while the UI is still very much usable, its usability was reduced by more than I would have liked. To actually provide good usability, a complete rework of the UI would be necessary. But this fell outside of the scope of the thesis.

In addition to the identified remaining issues, there might well be further, undiscovered issues related to IDE cross-compatibility. Since many such issues are caused by slight technical differences in the implementations of the different IDEs, they are very hard to discover purely through code analysis. But, even if such issues should present themselves in the future, they will most likely be of a smaller scale, requiring far less work than the issues resolved as part of this thesis.

In addition to resolving issues related to IDE cross-compatibility, I simplified and improved many Saros interfaces along with their implementations and documentation related to the Saros filesystem and sharing model. This significantly lowers the hurdle for adding Saros implementations for other IDEs, making it easier to further extend the reach of the Saros plugin. As a result of supporting more IDEs, the benefit of supporting IDE cross-compatibility would increase as well.

16 Future Work

Following this thesis, the obvious next step would be extending the STF to cover the new cases introduced through the new sharing model as described in issue #1045.

For near-future improvements, I would suggest resolving the remaining roadblock and the other issues described in section 14. Furthermore, I would suggest doing more in-depth IDE cross-compatibility testing to look for additional, more subtle issues.

For longer-term goals, I would suggest working on a complete rewrite of the current UI for the different Saros implementations to better support the new, reference-point-based sharing model. During this rework, the issues described in section 14.2 should be kept in mind. Additionally, I would suggest completing the Saros implementation for IntelliJ IDEA as well as providing Saros implementations for a broader range of IDEs to maximize the potential user base and range of use cases.

Bibliography

[1] A. Cockburn and L. Williams, "The Costs and Benefits of Pair Programming," in *eXtreme Programming and Flexible Processes in Software Engineering XP2000*, pp. 223–247, Addison-Wesley, 2000.

[2] B. Böckeler and N. Siessegger, "On pair programming." https://martinfowler.com/articles/on-pair-programming.html, Jan. 2020.

[3] J. Schenk, *Industrially Usable Distributed Pair Programming*. PhD thesis, Freie Universität Berlin, Inst. für Informatik, 2018.

[4] R. Djemili, "Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung," diploma thesis, Freie Universität Berlin, Inst. für Informatik, 2006.

[5] S. Rossbach, "Einführung einer kontinuierlichen Integrationsumgebung und Verbesserung des Test-Frameworks," bachelor's thesis, Freie Universität Berlin, Inst. für Informatik, 2011.

[6] A. E. Solovjev, "Operationalizing the Architecture of an Agile Software Project," master's thesis, Freie Universität Berlin, Inst. für Informatik, 2014.

[7] T. Bouschen, "Bestandsaufnahme und Arbeit an einer Alpha-Version des Saros-Plugins für die IntelliJ-Plattform," bachelor's thesis, Freie Universität Berlin, Inst. für Informatik, 2017.

[8] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping, "High-latency, low-bandwidth windowing in the jupiter collaboration system," in *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, (New York, NY, USA), p. 111–120, Association for Computing Machinery, 1995.

[9] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 5, p. 63–108, Mar. 1998.

[10] A. Lasarzik, "Refaktorisierung des Eclipse-Plugins Saros für die Portierung auf andere IDEs," bachelor's thesis, Freie Universität Berlin, Inst. für Informatik, 2015.

[11] D. Washington, "Entwicklung und Evaluation eines unabhängigen Sitzungsservers für das Saros-Projekt," master's thesis, Freie Universität Berlin, Inst. für Informatik, 2016.

[12] D. Sungaila, "Verbesserung und Erweiterung der Core-Bestandteile von Saros," bachelor's thesis, Freie Universität Berlin, Inst. für Informatik, 2016.

[13] O. Hanßen, "Saros: Refactoring the Filesystem," master's thesis, Freie Universität Berlin, Inst. für Informatik, 2019.

[14] L. Prechelt, "Some non-usage data for a distributed editor: The saros outreach," in *Proceedings of the 4th International Workshop on Cooperative and Human Aspects*

of Software Engineering, CHASE '11, (New York, NY, USA), p. 48, Association for Computing Machinery, 2011.

[15] L. Prechelt and K. Beecher, "Four generic issues for tools-as-plugins illustrated by the distributed editor saros," in *Proceedings of the 1st Workshop on Developing Tools as Plug-Ins*, TOPI '11, (New York, NY, USA), p. 9–11, Association for Computing Machinery, 2011.

[16] E. Rosen, S. Salinger, and C. Oezbek, "Project kick-off with distributed pair programming," in *PPIG*, 2010.

[17] S. Salinger, C. Oezbek, K. Beecher, and J. Schenk, "Saros: An eclipse plug-in for distributed party programming," in *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, (New York, NY, USA), p. 48–55, Association for Computing Machinery, 2010.

V Appendix

A Pull Requests

#842 *Fix handling of non-text editors*
https://github.com/saros-project/saros/pull/842

#860 *[INTERNAL][CORE] Remove text edit listener from FollowModeManager*
https://github.com/saros-project/saros/pull/860

#862 *Refactor selection handling*
https://github.com/saros-project/saros/pull/862

#868 *[FIX][CORE] Fix wrong split operation combination*
https://github.com/saros-project/saros/pull/868

#872 *Refactor text activity position handling*
https://github.com/saros-project/saros/pull/872

#874 *Add inclusion transformation fuzzer*
https://github.com/saros-project/saros/pull/874

#877 *Introduce text edit content normalization*
https://github.com/saros-project/saros/pull/877

#878 *Adjust consistency watchdog to work with normalized content*
https://github.com/saros-project/saros/pull/878

#880 *[UI][E] Remove access to partial sharing feature*
https://github.com/saros-project/saros/pull/880

#882 *Remove partial sharing logic*
https://github.com/saros-project/saros/pull/882

#886 *Ignore unknown SuppressWarnings arguments in Eclipse*
https://github.com/saros-project/saros/pull/886

#889 *Fix minor issue with resource activity filter logic*
https://github.com/saros-project/saros/pull/889

#898 *Clean up file list factory*
https://github.com/saros-project/saros/pull/898

#900 *Fix text activity offset handling*
https://github.com/saros-project/saros/pull/900

#905 *[INTERNAL][I] Use content for file move activities*
https://github.com/saros-project/saros/pull/905

#906 *Remove IResource.move(...) method*
https://github.com/saros-project/saros/pull/906

#907 *Clean up resource interface*
https://github.com/saros-project/saros/pull/907

#908 *Replace resource type list with enum*
https://github.com/saros-project/saros/pull/908

#913 *Improve charset handling*
https://github.com/saros-project/saros/pull/913

#914 *Require charset for file activity containing binary content*
https://github.com/saros-project/saros/pull/914

#915 *Remove IContainer.getDefaultCharset()*
https://github.com/saros-project/saros/pull/915

#916 *Remove IResource.adaptTo(Class)*
https://github.com/saros-project/saros/pull/916

#918 *[NOP] Remove FolderMoveActivity*
https://github.com/saros-project/saros/pull/918

#930 *[FIX][I] #924 React after document modification to avoid cache issues*
https://github.com/saros-project/saros/pull/930

#934 *Improve IResourceActivity resource abstraction*
https://github.com/saros-project/saros/pull/934

#938 *Remove SPath*
https://github.com/saros-project/saros/pull/938

#943 *Remove IProject.findMember(IPath)*
https://github.com/saros-project/saros/pull/943

#950 *[INTERNAL] Remove IWorkspaceRoot*
https://github.com/saros-project/saros/pull/950

#952 *Move remaining methods from IProject to IContainer*
https://github.com/saros-project/saros/pull/952

#953 *Consolidate IResource path handling*
https://github.com/saros-project/saros/pull/953

#970 *[BUILD][E] Export package 'saros.filesystem.checksum' from core*
https://github.com/saros-project/saros/pull/970

#976 *Clean up IWorkspace*
https://github.com/saros-project/saros/pull/976

#977 *Add javadoc to resource interfaces*
https://github.com/saros-project/saros/pull/977

#978 *Add default resource methods*
https://github.com/saros-project/saros/pull/978

#981 *[FIX][CORE] Fix handling of empty selection in text selection activity*
https://github.com/saros-project/saros/pull/981

#995 *[API][CORE] Adjust IContainer.getFolder(...) to not allow empty paths*
https://github.com/saros-project/saros/pull/995

#997 *Introduce reference point resources (ref-point migration 1/5)*
https://github.com/saros-project/saros/pull/997

#998 *Migrate inner classes to reference points (ref-point migration 2/5)*
https://github.com/saros-project/saros/pull/998

#999 *Migrate UI classes to reference points (ref-point migration 3/5)*
https://github.com/saros-project/saros/pull/999

#1000 *Clean up after migration to reference points (ref-point migration 4/5)*
https://github.com/saros-project/saros/pull/1000

#1001 *Migrate local resource handler to reference points (ref-point migration 5/5)*
https://github.com/saros-project/saros/pull/1001

#1005 *[API][CORE] Return list for IContainer.members()*
https://github.com/saros-project/saros/pull/1005

#1008 *Remove reference point mentions in UI*
https://github.com/saros-project/saros/pull/1008

#1012 *Clean up for core after reference point migration*
https://github.com/saros-project/saros/pull/1012

#1023 *[INTERNAL][E] Introduce reference point based resource implementation*
https://github.com/saros-project/saros/pull/1023

#1026 *Set up Saros/E remote debugging for IntelliJ IDEA*
https://github.com/saros-project/saros/pull/1026

#1032 *Migrate internal classes to use new resource implementation (ref-point migration 1/10)*
https://github.com/saros-project/saros/pull/1032

#1033 *Migrate filesystem handlers to use new resource implementation (ref-point migration 2/10)*
https://github.com/saros-project/saros/pull/1033

#1034 *Migrating ui internals to new resource implementation 1/3 (ref-point migration 3/10)*
https://github.com/saros-project/saros/pull/1034

#1035 *Migrate sender resource selection logic to use new resource implementation (ref-point migration 4/10)*
https://github.com/saros-project/saros/pull/1035

#1036 *Migrating ui internals to new resource implementation 2/3 (ref-point migration 5/10)*
https://github.com/saros-project/saros/pull/1036

#1037 *Migrate receiver resource selection logic to use new resource implementation (ref-point migration 6/10)*
https://github.com/saros-project/saros/pull/1037

#1038 *Clean up after resource implementation migration 1/2 (ref-point migration 7/10)*
https://github.com/saros-project/saros/pull/1038

#1039 *Migrating ui internals to new resource implementation 3/3 (ref-point migration 8/10)*
https://github.com/saros-project/saros/pull/1039

#1040 *Fix STF after migration to new resource implementation (ref-point migration 9/10)*
https://github.com/saros-project/saros/pull/1040

#1049 *Clean up after resource implementation migration 2/2 (ref-point migration 10/10)*
https://github.com/saros-project/saros/pull/1049

#1051 *Improve local reference point mapping suggestion*
https://github.com/saros-project/saros/pull/1051

#1058 *Add sanity check to prevent nested reference points*
https://github.com/saros-project/saros/pull/1058

#1061 *[INTERNAL][CORE] Pass JID of local user to session CTOR*
https://github.com/saros-project/saros/pull/1061

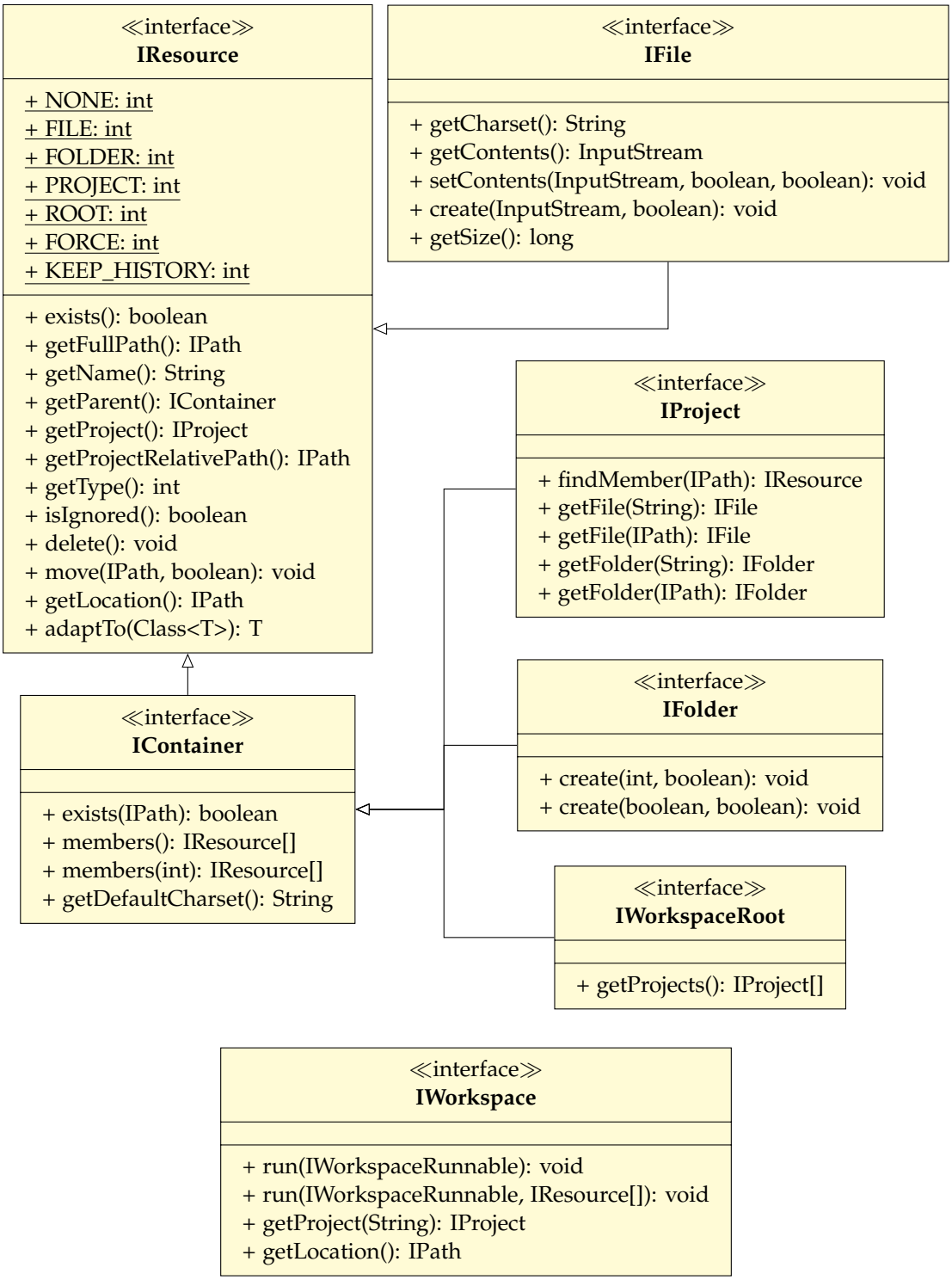
#1072 *Move checksum cache into session context*
https://github.com/saros-project/saros/pull/1072

#1085 *[DOC] Remove migration to reference point system as an open topic*
https://github.com/saros-project/saros/pull/1085

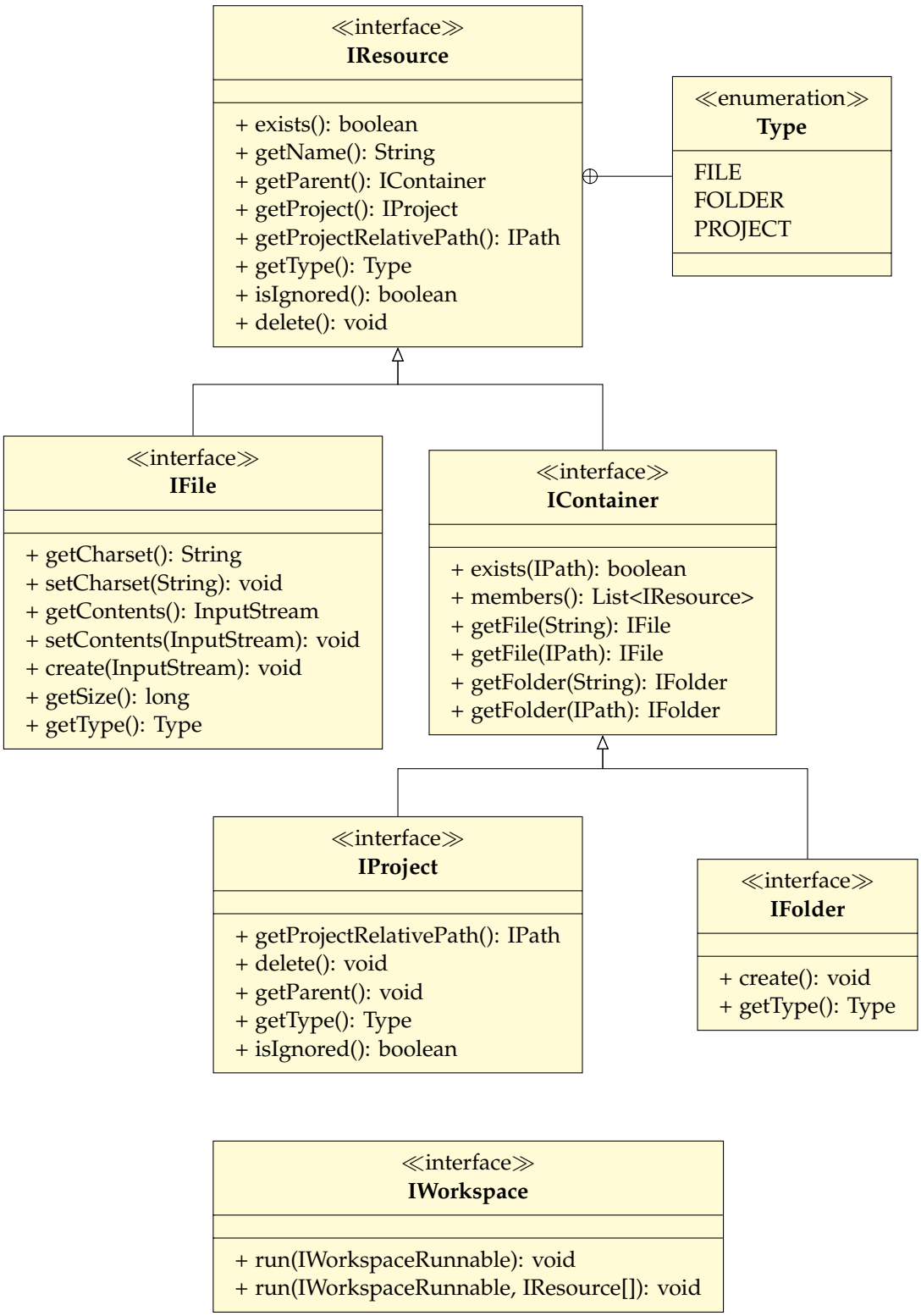
#1097 *[NOP][CORE] Remove the ROOT resource type*
https://github.com/saros-project/saros/pull/1097

B Figures

B.1 UML Diagram before and after the Filesystem Cleanup



(a) Filesystem interfaces before the cleanup.



(b) Filesystem interfaces after the cleanup.

Figure B.1: The filesystem interfaces (a) before and (b) after the cleanup.